Opportunistic Data Flow Integrity for Real-time Cyber-physical Systems Using Worst Case Execution Time Reservation

Yujie Wang, Ao Li, Jinwen Wang, Sanjoy Baruah, Ning Zhang

Washington University in St. Louis

Abstract

With the proliferation of safety-critical real-time systems in our daily life, it is imperative that their security is protected to guarantee their functionalities. To this end, one of the most powerful modern security primitives is the enforcement of data flow integrity. However, the run-time overhead can be prohibitive for real-time cyber-physical systems. On the other hand, due to strong safety requirements on such real-time cyber-physical systems, platforms are often designed with enough reservation such that the system remains real-time even if it is experiencing the worst-case execution time. We conducted a measurement study on eight popular CPS systems and found the worst-case execution time is often at least five times the average run time.

In this paper, we propose opportunistic data flow integrity, OP-DFI, that takes advantage of the system reservation to enforce data flow integrity to the CPS software. To avoid impacting the real-time property, OP-DFI tackles the challenge of slack estimation and run-time policy swapping to take advantage of the extra time in the system opportunistically. To ensure the security protection remains coherent, OP-DFI leverages in-line reference monitors and hardware-assisted features to perform dynamic fine-grained sandboxing. We evaluated OP-DFI on eight real-time CPS. With a worst-case execution time overhead of 2.7%, OP-DFI effectively performs DFI checking on 95.5% of all memory operations and 99.3% of safety-critical control-related memory operations on average.

1 Introduction

Real-time embedded systems play a critical role in modern society, as they are used to control and monitor various safety-critical processes, such as air traffic control, medical operations, and autonomous driving. Besides real-time responsiveness, security of these systems is also essential for safety, where it is widely acknowledged that there is no safety without security [46].

However, due to the constraints on processor speed, the trade-off between performance and security has been a major obstacle in embedded system designs. As a result, to maintain form factor and low cost, security overhead often has to be very small [65]. Overhead from data flow integrity (DFI) ranges from 100% to 200% is often considered prohibitive for most real-time embedded systems.

Real-time Scheduling and Security Opportunities: While there will always be a trade-off between security and performance, there are unique properties in real-time cyber-physical systems offering new opportunities. One key observation comes from the difference between the average execution time and the worst-case execution time (WCET). In real-time systems, the most important property is that all tasks need to complete in a timely manner with no task exceeding its deadline, since there is risk in physical world consequences (such as a plane crashing or a collision with pedestrians). In order to ensure this property, real-time system designers perform schedulability analysis using the WCET of the tasks to ensure all the tasks can complete even in the worst case. However, as the worst case is rare most of the time, the typical runtime of a task is significantly less than its WCET. In our preliminary analysis of eight CPS platforms, the worst case is more than five times of the average case for many tasks. Such difference is often referred to as *slack* in real-time system literature. As a result, after the real-time task finishes its execution, the system often remains idle. Recognizing this opportunity, existing work has explored utilizing such time for additional system checks [25,41] or additional non-critical tasks in mix-criticality systems [36].

OP-DFI: Building on top of the observation that slack opens up new opportunities to improve system security opportunistically, we propose OP-DFI, which takes advantage of slack to perform data flow integrity checks. This allows the deployment of computationally intensive data-flow integrity on real-time embedded platforms without impacting the real-time requirement. Different from existing works that leverage opportunistic execution [25] to perform memory checksum

verification, the targeted security primitive dictates that OP-DFI cannot wait till the task completion and has to be executed as in-line reference monitors for data flow integrity, presenting a fundamentally different set of challenges to meet both the real-time and security requirements.

Technical Challenges and Solutions: The challenges in OP-DFI arise from the need to accurately estimate the amount of slack, formulate a security policy leveraging such slack, and enforce the security policy in runtime.

Challenge 1: Slack Estimation. The key idea behind our slack estimation approach is to leverage program inputs and contexts to eliminate infeasible worst-case execution path (WCEP) for better estimation of slack. However, enumerating all the paths to recalculate the new WCET imposes prohibitive runtime overhead. To address this, path constraints and new WCET are pre-computed and stored in a table. Yet the table can be very large, imposing prohibitive memory overhead. To tackle this, OP-DFI leverages only a small number of constraints at the cost of further under-approximation of slack. To maximize the impact of the limited constraint evaluations, the selection of constraints takes advantage of the predictable nature of CPS and optimizes over the expected input distribution. Furthermore, since slack estimation is under-approximated to ensure real-time requirements are met, this may result in a lack of computation time earlier in the task execution. To tackle this issue, we make the observation that if critical data is isolated, as long as it has not been written, it does not have to be tracked. To minimize runtime overhead, OP-DFI makes use of memory tag extension (MTE) in ARM hardware to perform the hardware-enforced isolation to prevent arbitrary data write, such that it remains possible to turn on DFI for more of these variables. Three sandboxes are maintained, the unprotected variables, the protected variables, and metadata.

Challenge 2: Translating Slack to Security Protection. Effective use of slack for data flow protection requires addressing several technical problems. First, to maximize the security protection of slack, the security policy has to balance between the probability of memory access and the cyber-physical security implications of the memory location to decide if it should be included. At the same time, OP-DFI also has to ensure that the constructed security policy does not violate real-time constraints by estimating its impact to the worst-case execution time. Lastly, OP-DFI also has to ensure that the selected data is indeed protected while leaving the rest of the program unchecked. To tackle this, OP-DFI formulates two properties of security coherence, from both spatial and temporal dimensions. The properties are based on the observation that the data corruption of protected variables can only be achieved by either tampering with their dependencies or by modifying them during the time window when they are not protected.

Challenge 3: Runtime Enforcement of Dynamic Policy. Due to changes in security policy over data flow during runtime, the reference monitor must adapt to new sets of data access rules by using different in-line reference monitors. While conceptually simple, implementing this in the system is very challenging, as the security policy and its enforcement are often accomplished via reference monitors in the form of software instrumentation. Updating the policy would modify the program instrumentation, or the reference monitor needs to have sufficient logic to switch between different security policies. Additionally, dynamically self-modifying the binary is not only problematic from the security perspective but also degrades performance due to cache coherence. To address this problem, OP-DFI trades the space for performance. More specifically, OP-DFI keeps different versions of the same code logic but with different levels of policy protection, and dynamically switches to the code with the desired policy.

Evaluation and Outcome: We implement a prototype of OP-DFI to enforce opportunistic DFI on the AArch64 platform. The prototype includes a slack analyzer for input-based slack estimation, a library for runtime enforcement, a customized compiler for automatic program instrumentation, and an optimization engine for optimal security policy generation. We evaluate the prototype on eight real-time CPS platforms. Compared to regular DFI with 184% WCET expansion, OP-DFI with 2.7% overhead can effectively perform DFI checking on 95.5% of all memory operations and 99.3% of safety-critical control-related memory operations on average. In summary, our contributions are:

- We propose a new security primitive, opportunistic DFI, capable of enforcing data-flow integrity by taking advantage of processor reservation for meeting the worst-case execution time of tasks.
- We tackle several technical challenges to design opportunistic DFI, including runtime estimation of slack, balancing security and real-time in slack utilization, as well as runtime enforcement of dynamic policy.
- We implement a prototype[1] and evaluate it on eight real-time CPS platforms to demonstrate its effectiveness.

2 Background

2.1 Real-time System

Real-time systems execute tasks with precise timing constraints [59–61]. In real-time CPSs [63], tasks are often recurring or periodic workloads, each instance of which must complete by a *deadline*, after which a delayed control signal may cause catastrophic results (e.g., a self-driving car crash). A formal guarantee of timeliness is achieved through *schedulability analysis*, which determines whether each instance of every task can complete within its deadline on the target system.

[1] Source code is available at https://github.com/WUSTL-CSPL/OP-DFI

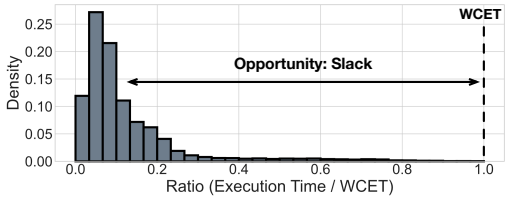


Figure 1: Opportunity in WCET reservation.

To this end, each task's workload is characterized according to its *worst-case execution time* (WCET). Because program execution paths may be input-dependent [37], a common approach to measuring WCET involves statically unrolling loops to derive the longest execution paths [42]. In combination with low-level timing data for basic blocks or functions, each path's execution time may be characterized. The path with the longest execution time, or *worst-case execution path* (WCEP), defines the WCET. Analysis techniques, such as utilization bound tests [39] and response time analysis [31], can then be used to determine the schedulability of a complete task system under the assumption of worst-case execution.

However, individual instances of a task often execute for less than the WCET. This discrepancy is referred to as *dynamic slack*. It can be regarded as the extra computational resource remaining after executing a task instance and can be used on other computations such as security checking [41].

2.2 Software Security

Data Flow Integrity: DFI aims to check whether loaded data is only modified by expected instructions at runtime [18]. Specifically, a software-based DFI mechanism instruments a program to taint bytes written to memory with the corresponding store instruction's ID assigned during compile time. Before any data is loaded from memory at runtime, the reference monitor checks whether the data has been modified by the set of expected instructions, which is obtained through compile-time static analysis.

ARM Memory Tag Extension: ARM Memory Tagging Extension (MTE) [1] is introduced by ARM on the AArch64 platform to provide defense against attacks that try to subvert code to process malicious data. MTE introduces two kinds of 4-bit value tags: address and memory. An *address tag* is stored at the top of each pointer. A *memory tag* is associated with every 16-byte memory region. The address and memory tags will be verified every time an instruction loads or stores data. If two tags are mismatched, a hardware fault will be triggered. Current Linux kernels support configuring whether MTE is enabled on a specific region of memory [2]. MTE also offers user-space instructions for manipulating memory tags. For instance, the instruction `STGP` is a variant of the regular store instruction `STR`. Unlike `STR`, `STGP` stores both the tag and data in memory.

ARM Branch Target Identification: Branch target identification (BTI) aims to mitigate jump-oriented programming (JOP) attacks [5]. With BTI, the targets of indirect branch instructions are enforced to be designated locations, such as the beginning of functions. Therefore, BTI can also be regarded as an efficient but coarse-grained control flow integrity (CFI).

Table 1: WCET reservation on real-time systems

Type	Platform	Hw	Task	# Traces	Ratio (%) (Ave. / WCET)
Drone	PX4 [45]	A53	altitute control	1M	13% (36/275 us)
			navigate	1M	15% (22/141 ms)
	Ardupilot [9]	A53	throttle loop	1M	7.8% (4.0/52.4 us)
			bearing update	1M	5.1% (2.2/43 us)
AV	Turtlebot [53]	A72	laser receive	100K	5.3%(10/187 ms)
			dwa planning	100K	39% (402/1016 ms)
	Autoware [33]	A78	path planning	100K	3.2% (90/2736 ms)
			mission planning	1M	9.9% (28/281 us)
	Jackal [30]	A78	laser handle	1M	14% (109/778 us)
			odometry handle	1M	8.4% (4.6/54.6 ms)
Robot	OpenMani. [48]	A53	trajectory make	3K	28% (4.1/14.2 ms)
			main control	1M	21% (15/70 ms)
	Unitree [54]	A57	fsm control	1M	22% (835/3697 us)
			trottling control	1M	13% (209/1512 us)
	OP3 [48]	A57	walking control	1M	17% (2.8/15.8 ms)
			head control	1M	1.7% (92/5259 us)

3 Motivation

Overhead of Direct Application of DFI: DFI is one of the most powerful tools to defend against advanced attacks, such as data-oriented attacks [17,19]. However, existing DFI implementations often impose an average overhead ranging from 100% to 200% to the program runtime [13, 18]. As a matter of fact, on the eight CPS platforms we studied, DFI leads to an average WCET increase of 184%. To provide a concrete example, *throttle loop* is a safety-critical real-time task within Ardupilot, where full DFI leads to runtime expansion of 125% for WCET. According to the task model from the Ardupilot documentation, the system is no longer schedulable when the overhead is greater than 44%.

Opportunity in Worst Case Execution Time Reservation: While direct application of DFI incurs prohibitive runtime overhead, the difference between WCET and average runtime does provide unique opportunities. To gain a more quantitative understanding of available slack in common safety-critical tasks in existing CPS platforms, we have conducted a measurement study over 16 real-time tasks supporting different CPS functions (navigation, throttle, trajectory, and more) selected from eight most widely studied CPS software across three types of CPS platforms (including drones, autonomous vehicles (AV), and robots) using four different types of processors (ARM A53, A57, A72, and A78). Details of our evaluation setup can be found in Table 1.

Measurement Setup - For each CPS, we select officially released missions to initiate its execution. For drones and autonomous vehicles, the missions involve following a predefined trajectory. For robots, their missions consist of a series of actions, such as standing up, walking, turning around, etc. More details about the concrete set of missions can be found in Appendix C.1. During these missions, we record the ex-

ecution time of two critical tasks. Most tasks are measured and calculated over 1 million loops. Four tasks have fewer traces due to their relatively low task release frequency. The WCET is obtained via aiT TimeWeaver [6], a state-of-the-art timing analysis tool used in the industry. To obtain the WCET of a task, aiT TimeWeaver first unrolls the task's control-flow graph (CFG) into paths based on loop bound information. The tool then leverages Arm CoreSight debugging feature to profile the fine-grained execution time of individual basic blocks. Based on these fragmented timing profiles, it calculates the execution time of the longest path and reports it as WCET.

Observations - As shown in Table 1, the average execution time for different tasks ranges from 1.7% to 39% of WCET, with an average runtime of just 13.9% of the WCET. To understand this from a statistical perspective, Figure 1 shows the distribution of the percentage of task execution time in relation to task WCET. As shown in the figure, 90% of execution time is less than 20% of the WCET. As a result, for a large percentage of the executions, there are ample opportunities to leverage slack to improve the security of the system.

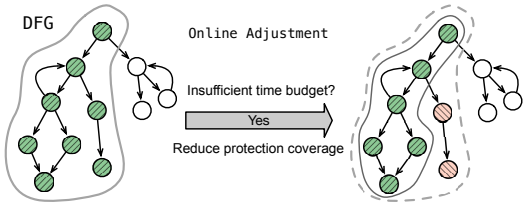


Figure 2: Workflow of opportunistic DFI.

Opportunistic DFI: Due to the large difference between WCET and the average runtime, it leaves a surplus of reserved time in most execution iterations, i.e., slack. However, direct application of existing data flow security primitives also leads to expansion of WCET. To capitalize on this opportunity, we propose a new security primitive, opportunistic DFI, which deploys DFI to perform data-flow checking whenever there is slack. Figure 2 shows the workflow of opportunistic DFI. Specifically, in-line reference monitors, which DFI uses for security checking, are deployed only on a portion of the program such that only a portion of data flows are checked. During execution, slack is monitored to adjust the coverage of DFI. For instance, if the slack is insufficient for the current DFI deployment, it reduces the amount of data flow to check. While the idea is simple, the design has to balance two requirements, timing and security respectively. From a timing perspective, WCET has to remain unaltered. From a security perspective, protection over the data has to be sound when only applied opportunistically over a subset of it. Details of the design are presented in Section 5.

4 Threat Model and System Goal

Threat Model: In this work, we focus on preventing memory corruption, therefore, we follow the common threat model of other DFI works [13, 18, 50, 51], trying to defend against data-oriented attacks [4, 26, 27, 29]. We assume there exists a memory corruption bug in the application software that can be exploited by attackers to read and write memory arbitrarily [3]. Consistent with other works [13, 18, 50, 51], we assume that code injection is prevented using existing memory protection technology such as MMU or MPU, therefore instrumentation cannot be tampered by the attacker. We also make the assumption that it is possible to prevent the instrumentation from being bypassed. This can be realized using software-based CFI implementations [34, 55] or hardware-based methods, such as the BTI feature in ARM [5]. Our design assumes that it's possible to sandbox the program. This sandboxing can be implemented using software-based fault isolation [57] or hardware-based mechanisms, such as ARM MTE [1]. Our implementation leverages the hardware features, ARM BTI and MTE, to enhance efficiency. Additional discussions on how to realize the CFI and sandboxing using a software-only approach are available in Section 11. Although the combination of CFI and sandboxing, such as BTI and MTE, provides powerful protection, the limited granularity of the sandbox still exposes a non-trivial attack surface for data-oriented attacks [44]. Our focus is memory-corruption-based attacks, therefore, hardware attacks [22], side-channel attacks [64], and performance interference attacks [38] are out of our scope. In this work, we focus exclusively on application security, assuming that all privileged software stacks, including the OS, hypervisor, and device firmware, are trusted. Lastly, though DFI is capable of preventing all illegal data flow (i.e data flows that do not follow the known DFG), the goal of OP-DFI is to provide as much data flow protection as possible without violating the real-time constraints, where the quantification of protection can take the form of either the number of read/write instructions or the number of critical variables or memory addresses.

5 OP-DFI Design

While the idea of utilizing slack for security is attractive, making use of such time without impacting the real-time system presents several unique challenges, from estimating the available slack during the execution of the program, to how to apply the slack to ensure data-flow integrity for critical data while maintaining the policy coherence. The system overview is shown in Figure 3.

5.1 On-the-fly Slack Estimation

Slack is defined by the difference between the time it takes to execute a task and its WCET. Such slack is easy to calculate upon the completion of the task, but it is impossible to obtain prior to the completion of the task. On the other hand, to make use of the time for security, the necessary condition is to have an under-approximation of the slack available, such

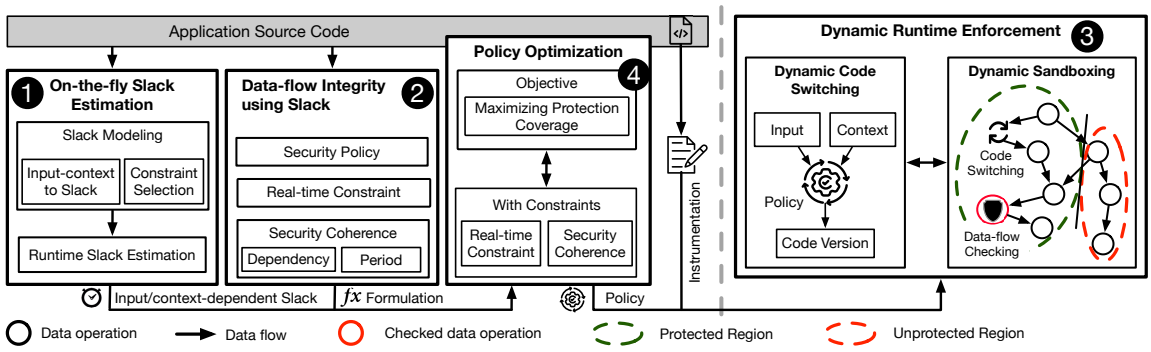


Figure 3: OP-DFI system overview.

that the system timeliness will not be impacted. To do so, we build on top of the observation that the execution time of a task depends on the program inputs (including sensor inputs, external messages, and program memory), assuming performance interference from other processes in the system has already been taken into consideration for the timing profile in the original design. While a subset of inputs are known at the start of the task (referred to as *input*), others will only become available during the execution of the program (referred to as *context*). The key idea behind run-time slack estimation in OP-DFI is that with the input and the known subset of context, then it is possible to prove some of the program paths to be infeasible. If the original WCEP happens to be in the set of infeasible paths, then WCET can be updated to provide a better estimate of the slack.

From Input-context to Slack: OP-DFI first employs symbolic execution to gather symbolic formulas for path constraints (i.e., branch conditions on paths). A subset of the formulas are computed at runtime using both the input and the known subset of context. The result is then considered as an estimate of the execution state. Since the collected formula might only be partially evaluated due to missing context, it is recalculated when more context becomes available, enabling a more accurate state estimation. With the estimated state as an index, the corresponding slack can be looked up from a pre-computed table, which is generated using Algorithm 1. Specifically, for an estimated state, Algorithm 1 checks the feasibility of all paths in the order from longest to shortest, continuing until it encounters a path that cannot be negated. The corresponding slack of that path is then taken as the estimated slack. This estimated slack is an under-approximation of the slack under the current state, as proved in Appendix A.1.

Constraint Selection: Conceptually, contexts can be used to prove that certain paths are infeasible, leading to a better estimate of slack. Therefore, a naive approach would be to insert checkpoints for new slack estimation at every single memory update relevant to the constraint formula. However, directly evaluating the full constraints over all paths incurs significant runtime overhead. An alternative is to encode them into lookup tables, but such a design leads to significant memory overhead and still incurs a non-trivial computational overhead

Algorithm 1: Parametric Estimation of Slack

Input: S_Ω // constraints eval results
Output: $\mathcal{B}$ // Worst-case slack estimate
// From long to short path
foreach p_i ***in*** $Sort(\{p_1, p_2, ...\})$ **do**
  if $Infea(p_i, S_\Omega)$ **then**
    continue // Path eliminated
  else
    return $WCET - t_i$ // Path's slack estimate

Note: S_Ω: { $(\omega_0 : r_0)$, $(\omega_1 : r_1)$, ... }, where r_i is the evaluation result for symbolic formula of branch constraints ω_i, and can be true, false or unknown (for missing context). $Infea(p, S_\Omega)$: true if the path condition of p violates S_Ω.

to calculate the table index from memory contexts. As a result, only a subset of constraints can be used. This leads to the problem of which subset should be selected will offer the higher probability of obtaining a better estimate of slack.

A straightforward method to balance overhead is to randomly select a limited number of constraints. However, not all constraints are equally effective. Given that the inputs for CPS typically follow specific distributions, certain path constraints are more frequently effective for path elimination in Algorithm 1. Based on this observation, we employ dynamic profiling to capture the mission-level CPS input distributions, which guides the optimization engine in selecting the constraints that are most effective for the most commonly appearing inputs. The details of the optimization formulation are in Section 5.4. Using this set of constraints, the expected gain in the bounds of slack is optimized over the expected input distribution. This, also in turn, enables more fine-grained code switching and enhances protection. Empirical measurements from our evaluation in Section 8.2 show the proposed method improves slack estimation accuracy by 15.3% on average.

5.2 Data-flow Integrity using Slack

Once an under-approximation of the slack is obtained, it is important to consider how this slack can be used for security. Unlike the conventional data-flow integrity, due to the limited slack, only part of the system can be protected (referred to as

the *protection region* and denoted by θ). Yet this leads to three challenges: 1) From the perspective of security policy, how to select which memory access instructions to instrument, 2) From the timeliness perspective, how to ensure the computational cost of the protection remains within the available slack, and 3) From the perspective of coherency for the protection, how to ensure that the selected data is indeed protected while leaving the rest of the program unchecked?

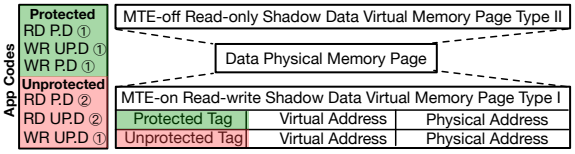
Security Policy: OP-DFI adopts a security policy π to select the protection target at runtime. The form of π is a mapping function that outputs a set of checked memory operations, θ, based on the estimated execution state S_Ω. This state is calculated using the current input and context, as discussed in Section 5.1. In synthesizing such a policy, there are two factors that govern the selection of memory operations. The first factor is variable usage. Variables not expected to be accessed before the next slack estimation checkpoint are directly included since they do not impact on runtime cost under normal conditions. For variables with uncertain usage, OP-DFI estimates access probability based on input data distribution and prioritizes the variables with higher probabilities. The second factor considers the importance of a variable's security implications, which can be derived from both cyber and physical perspectives. From the cyber aspect, variables that are more depended upon by other variables may be prioritized due to their broader impact. Regarding the physical aspect, variables associated with critical physical meanings could be prioritized. For example, control variables essential to the cyber-physical loop may receive higher priority for protection. To facilitate the selection from physical aspect, OP-DFI enables users to define policies that specify the criticality of variables. These factors are incorporated as weights in the synthesis of the security policy π, details of which will be provided in Section 5.4.

Real-time Constraint: OP-DFI imposes the constraint that, for every possible estimated state S_Ω, the computational cost of the checked memory operations must not exceed its estimated slack. Formally, the constraint is:

$$\forall S_\Omega, \Delta(S_\Omega, \pi) \leq \mathcal{B}(S_\Omega) \tag{1}$$

where $\mathcal{B}(S_\Omega)$ is the slack estimate obtained in Section 5.1. $\Delta(S_\Omega, \pi)$ represents the runtime cost of the checked memory operations, which is overestimated using the highest cost among all potential paths for the estimated state S_Ω.

For each individual potential path associated with the estimated state S_Ω, we quantify the runtime cost induced by a security policy in terms of the number of instructions added to the execution path by the policy. Given the security policy can be updated at each slack estimation checkpoint, the runtime cost of the security policy can be different between different checkpoints. To address this, OP-DFI divides the paths into subpaths at each slack estimation checkpoint and individually quantifies their runtime costs. The detailed derivation of $\Delta(S_\Omega, \pi)$ is discussed in Appendix A.2.



RD: Read, WR: Write, P.D: Protected Data, UP.D: Unprotected Data
RD/WR instructions' address operands are masked to one of two virtual pages

Figure 4: Dynamic sandboxing.

Security Coherence: Recognizing that the data corruption of protected variables can only be achieved by either tampering with their dependencies or by modifying them during the time window when they are not protected, there are two dimensions of security coherence that the policy needs to guarantee. The first dimension concerns the spatial aspect: given that attackers can compromise data through its dependencies, it follows that if a data item is included in the protected region, all its data and control dependencies must also be included. The second dimension focuses on the temporal aspect: given that the protected region can change at slack estimation checkpoints during runtime, attackers could exploit unprotected periods to compromise data. Since predicting the timing of attacks is challenging without additional security measures, effective protection can only be maintained if a variable remains protected throughout the task loop. In other words, the protected region can only shrink during runtime.

5.3 Dynamic Runtime Enforcement

As the protected region is consistently updated at slack estimation checkpoints, this leads to two challenges: 1) How to change the policy enforcement mechanism in the code instrumentation, since data-flow checking is generally enforced by in-line reference monitors via program instrumentation, which is not mutable at runtime. 2) How to ensure the protection remains coherent when protection boundary changes.

Dynamic Code Switching: To adapt the security policy at runtime, one direct method is to perform runtime binary rewriting. However, this approach can significantly degrade performance due to cache coherence issues. OP-DFI addresses this by maintaining multiple versions of code that have the same logic but different levels of protection. It then integrates a code-switching Reference Monitor (RM) into each slack estimation checkpoint to enable real-time switching between these code versions. For the generation of multiple code versions, an intuitive approach would be to create a unique code version for each estimated slack determined in Section 5.1. However, maintaining such a large number of code versions can lead to significant memory overhead. To mitigate this, we limit the number of code versions. Furthermore, when different code versions share the same code regions, we leverage MMU with address alignment to share these common code pages, reducing memory consumption.

To determine the switching target, the RM first re-evaluates the constraint formula using the input and context, and then

outputs the estimated state S_Ω. Subsequently, S_Ω is input into the security policy, which results in the identification of the transfer target. Conceptually, the target code version is usually the one that can provide the highest level of protection within the updated slack budget. Detailed implementation of code switching RMs are in Section 6.

Dynamic Sandboxing: There are two requirements for the second challenge. First, upon boundary changes, the protected data should remain isolated from the unchecked memory operations. Second, the overhead on the boundary change process is minimized.

Sandboxing is a natural solution for achieving the isolation, but its static nature limits its applicability in our scenario. To address this, we propose a novel dynamic sandboxing approach, where the isolation boundary is dynamically adjusted based on changes in the protected region. However, directly implementing dynamic sandboxing is problematic. Traditional sandboxing relies on inline instruction masking to sanitize addresses. Dynamic changes of the protection boundary can incur prohibitive overhead, as they require moving data between sandboxes, thereby violating the second requirement. To handle this issue, our system leverages ARM MTE to facilitate access control. We assign distinct MTE tags to the checked and unchecked code regions. Every time a memory is modified, its memory tag is updated as the tag of the write instruction. Thus, any unauthorized tag write on protected data by unchecked code will eventually be detected during checked read operations, as the stored memory tag is inconsistent with the instruction tags. The advantages of ARM MTE is its support for the `STGP` instruction, which allows data and memory tags to be modified simultaneously. By replacing the `STR` instruction with `STGP`, the need for additional instructions to update memory tags upon protection boundary changes is eliminated, thereby avoiding extra overhead.

Unfortunately, the use of ARM MTE introduces a new challenge: it indistinguishably prevents both read and write operations on protected data from unchecked code. This limitation could disrupt the program's normal execution, as unchecked code may need to read the protected data. To mitigate this issue, we introduce a second virtual page with MTE-off and read-only privilege to enable access, as shown in Figure 4. Lastly, to prevent unchecked write operators from updating the metadata in MTE, an instrumentation is added to enforce a known "unprotected" mask for all those memory write instructions.

5.4 Opportunistic DFI Policy Optimization

From the previous sections, the effectiveness of a security policy is influenced by various factors, including the formation of protected memory operations, the choice of path constraints, and the selection of code versions for different slack. Besides, these factors are subject to multiple constraints, namely real-time constraint, security coherence, and limitations on memory impact. Our goal is to find a security policy that maximizes our security objectives without violating any of the three constraints. To achieve this, we naturally formulate it as an optimization problem.

Security Objective: To quantify the security objectives within the optimization framework, OP-DFI proposes a metric called *coverage score*. A naive approach for calculating this score is to simply count the number of checked memory operations, treating all operations as if they were of equal importance. As discussed in Section 5.2, the security implications of variables can vary, meaning that allocating more of the time budget to more important variables could enhance overall security. Thus, OP-DFI introduces weights for variables to indicate their importance. To identify such importance, sensitivity analysis [10, 62] is a potentially promising solution. To capture the difference in the importance, a weight is added to each operation in the problem formulation. With the weights, the protection coverage score $\mathcal{C}$ for a set of selected path constraints Ω_s and policy π is calculated as the weighted average percentage of checked memory operations:

$$\mathcal{C}(\Omega_s, \pi) = \sum_{x \in X} \Pr(x) * \Big(\sum_{\vartheta \in \pi(S_\Omega(x))} w_\vartheta \Big). \tag{2}$$

Where x represents the input and context, $\Pr(x)$ is the dynamically profiled distribution of x, and $S_\Omega(x)$ signifies the execution state estimated using x. The variable w_ϑ represents the importance weight assigned to a memory operation ϑ with $\sum_\vartheta w_\vartheta = 1$. By default, w_ϑ is set to $\frac{1}{N}$, where N is the total number of memory operations. Additionally, $\pi(S_\Omega(x))$ represents the final set of protected memory operations. It is important to note that the simple formulation above captures only memory operations but not how it relates to critical variable or memory coverage, which is an important direction for our future work.

Optimization: The optimization problem is formulated as finding $(\Omega_s, \pi)^*$ that maximizes $\mathcal{C}(\Omega_s, \pi)$, subject to a set of constraints on Ω_s and π. These constraints consist of limitations on the runtime and memory overhead imposed by Ω_s and π, as well as the requirements on π for security coherence. To solve the optimization problem, we employ a combined approach that leverages the genetic algorithm for its capability in solving non-linear problems, and mixed-integer linear programming for its efficiency. More details on the optimization problem formulation and solving can be found in Appendix A.3 and B.1 respectively.

Even though our evaluation across eight platforms in Section 8 didn't yield any instances where the optimization produced a null policy, a policy protects no memory operations, it remains possible that in extreme cases, the output policy is null. For example, when all variables are mutually dependent, a substantial amount of slack would be needed upfront to protect any individual variable due to its dependencies on others. In these situations, no suitable policies would be available, and OP-DFI would not offer any additional protection.

6 Implementation

A prototype of OP-DFI is built on AArch64, and the implementation mainly contains four parts, including slack analyzer, code switching RM, program instrumentation, and optimization engine. The policy optimization engine is described in Appendix B.1 while others are described below.

Slack Analyzer: To compute path-level slack, timing analysis [24] is conducted. This involves several steps: First, dynamic analysis is used to profile the timing characteristics of various basic blocks. Second, loops are unrolled based on the bounds given in the schedulability analysis of the system. Subsequently, path information and low-level timing data are combined to calculate slack using the industrial-standard timing analysis tool, aiT TimeWeaver [32]. To obtain symbolic formulas for constraints, we utilize a symbolic execution engine based on KLEE [16]. Given that a threshold is set to bound constraint evaluation cost, which also limits the complexity of the constraint formula, the depth of symbolic execution is limited as well, thereby preventing path explosion. With the chosen constraints, slack estimation checkpoints are added at relevant variable updates. Given that the constraints pertain to specific variable updates, each checkpoint uses an update counter to differentiate between multiple updates for the same variable.

Code Switching RM: For efficient determination of the switching target, the switching rules, automatically generated by the optimization engine, are implemented as a precomputed lookup table, where the estimated execution state is utilized as the index. Moreover, to prevent potential attackers from exploiting the system by maliciously jumping to code versions lacking protection, each code version is enclosed within a compartment, achieved by aligning its code addresses to ensure differences only exist in the high bits. A register, which can only be updated by reference monitors, is reserved to store the identifier of the current code version. This identifier is used to conduct address masking on each indirect branch, guaranteeing that the control flow remains confined within the correct compartments at all times.

Program Instrumentation: To enforce hard-coded MTE tags, masking instructions are inserted before both read and write instructions. Similar to the optimization techniques used in static sandboxing [49], repeated masking for registers used in frequent memory accesses, such as stack accesses, can be eliminated. To prevent attackers from directly jumping to the middle of the instrumented code, a coarse-grained CFI mechanism is employed as a fundamental security feature. Specifically, ARM BTI instructions are inserted at each function entry and potential function return address. The return address is further protected with a shadow stack, implemented using a separate MTE tag. Additionally, all metadata is protected with a distinct MTE tag.

7 Case Study

```
void generate_attitude_setpoint (const Quatf &q, ...){
    const float yaw = Eulerf(q).psi();
    _man_yaw_sp = yaw;
    /* . . . */
    // fly towards the direction of (roll, pitch)
    Vector2f v =  Vector2f(_m_control_setpoint.roll *
         _man_tilt_max), _m_control_setpoint.pitch *
         _man_tilt_max);
    Quatf q_sp_rp = AxisAnglef(v(0), v(1), 0.f);
    /* . . . */
    //calculate how much the yaw changes
    const Quatf q_sp_yaw(cosf(_man_yaw_sp / 2.f), 0.f,
         0.f, sinf(_man_yaw_sp / 2.f));
    correctTiltSetpointForYawError(q_sp_rp, q, q_sp_yaw);
     /* . . . */}
```

```
Variable Set #1
_m_control_setpoint.roll
_m_control_setpoint.pitch
q_sp_rp   _man_titlt_max
Variable Set #3
_man_yaw_sp   q_sp_yaw
yaw
Variable Set #2   q
```

Listing 1: Code snippet of the case study.

In this case study, OP-DFI is applied to the CPS task *altitude control* of PX4 [45]. Listing 1 shows code snippets taken from the altitude setpoint generation process in the altitude controller, with irrelevant code omitted for simplicity. This process involves a procedure to correct setpoints in the presence of a yaw error. The correction procedure takes three control variables: `q_sp_rp` (pure tilt quaternion that needs correction), `q` (current attitude), and `q_sp_yaw` (pure yaw quaternion). These variables are categorized into inclusive/exclusive sets based on their dependencies, in accordance with the security-coherence constraint. Subsequently, the optimization engine generates the optimal policy to maximize the quantified protection coverage score. The process of constructing the coverage scores is accomplished through Sensitivity Analysis [62]. The variables of configurations with the most substantial influence are given priority for dynamic protection. In our case, the control variable `q_sp_rp` in the altitude controller is identified as more critical than the control variables `q_sp_yaw` and `q` in the context of altitude setpoint corrections, and it is highlighted in red in Listing 1. As a result, memory operations on its dependent variables (set 1) are prioritized for protection by assigning higher optimization weights (ten times larger in our setting) than to other memory operations. With OP-DFI deployed, it achieves a 94.2% protection coverage with just a 2.8% increase in WCET. Compared to regular DFI, the WCET is reduced by 108%.

8 Evaluation

Our evaluation aims to answer the following questions: **(1)** What is the system overhead and provided protection of OP-DFI compared with full DFI? (Section 8.1) **(2)** What are the factors affecting slack estimation? (Section 8.2) **(3)** What is the trade-off between different code versions? (Section 8.3) **(4)** How effective is OP-DFI when facing an adaptive attacker? (Section 8.4).

Evaluation Setup: The evaluation is conducted on eight real-time CPS platforms [9, 30, 33, 45, 48, 53, 54], as shown in Table 2. Among the eight CPS, PX4, Ardupilot, Turtlebot, and OpenManipulator are evaluated in real-world scenarios, while the others are evaluated in simulation mode. We utilized all eight platforms for the evaluation question (1) (Section 8.1),

Table 2: Runtime performance of OP-DFI

CPS	Baseline (ms)				System Runtime Overhead (%)					Reduced WCET	Coverage Score	Protected Ctr.Var.
	Average Runtime		WCET		Average Runtime			WCET				
	w.o DFI	w. full DFI	w.o DFI	w full DFI	DF checking	code switching	sandboxing	code swiching	sandboxing			
PX4	22	48	141	311	116	2.82	6.1	0.32	2.21	118.0%	0.98	99.8%
Ardupilot	0.004	0.007	0.052	0.117	76	6.3	4.2	0.66	1.98	123.2%	0.93	99.6%
Turtlebot	10	26	187	490	153	5.82	5.2	0.42	2.39	159.2%	0.95	98.5%
Autoware	90	440	2736	12941	383	8.31	7.3	0.22	3.09	369.7%	0.99	99.7%
Jackal	0.109	0.219	0.778	2.372	97	2.1	6.4	0.47	2.04	202.4%	0.95	99.3%
OpenMani.	15	33	70	173	117	1.93	5.6	0.28	2.28	144.6%	0.97	99.4%
Unitree	0.835	1.829	3.697	8.546	99	3.8	4.4	0.57	2.33	128.3%	0.92	98.9%
OP3	2.8	7.1	15.8	48.6	142	2.44	4.7	0.34	2.67	204.6%	0.95	99.3%

w: with, w.o: without, DF: Data Flow, Ctr.Var.: Control Variable

which focuses on macroscopic performance metrics. Subsequently, to answer the evaluation questions (2) - (4) (from Section 8.2 to 8.4), we selected PX4, Turtlebot, and OpenManipulator. These platforms are selected for their real-world settings and represent CPS platforms for drones, autonomous vehicles, and robots, respectively. The evaluated tasks for each CPS (in the order listed in Table 2) are *navigate*, *throttle loop*, *laser receive*, *path planning*, *laser handle*, *main control*, *fsm control*, and *walking control*. The evaluated hardware is Cortex-A53 on Raspberry Pi 3 model B (PX4, Ardupilot, OpenManipulator), Cortex-A57 on Jetson Nano (OP3, Unitree), Cortex-A72 on Raspberry Pi 4 model B (Turtlebot) and Cortex-A78 on Jetson AGX Orin (Autoware, Jackal). For security policy generation, we set the number of code versions, the number of selected path constraints, and the threshold for constraint evaluation cost to 6, 16, and 0.3% of WCET, respectively. We prioritized the protection of control variable-related memory operations by assigning them optimization weights 10 times larger than others. However, for clarity in presenting protection coverage within the program, the coverage scores displayed in this section represent the average percentage of checked memory operations. Moreover, due to the lack of publicly available hardware for ARM MTE and BTI, instruction analogs are employed to assess performance. Regarding WCET measurement, we are utilizing the aiT TimeWeaver [6], a timing analysis tool. Further details regarding the instruction analogs and WCET measurement can be found in Appendix C.1.

8.1 Comparison with Full DFI

To answer evaluation question (1), we measured the offered protection and system overhead, including average runtime, WCET, and memory overhead, under the CPS missions as described in Table 7.

Baseline DFI: The baseline DFI [18] enforces inter-procedure data-flow checking using a read-write table, write-tainting, and read-checking instrumentation. Specifically, the read-write table keeps an entry for each 4-byte word. Each memory write is assigned a 2-byte label and is instrumented to update the table entry with its label when writing, and the memory read is instrumented to check if the memory content has been written by a legitimate memory write according to the result of static analysis.

Runtime Overhead: Both average and WCET overhead of OP-DFI are measured and shown in Table 2.

Average Runtime Overhead - OP-DFI results in an average runtime overhead of 157.5%, attributed to data-flow checking (147.8%), code switching (4.19%), and dynamic sandboxing (5.4%). Among these factors, data-flow checking in OP-DFI is comparable to full DFI (147.8% versus 155.3%) and accounts for the most significant portion of the average runtime overhead due to the abundantly available slack. The dynamic code switching mechanism incurs a runtime cost of 4.1%, attributable to the evaluation of constraint formulas, tracking runtime context, and executing transitions between code versions. Moreover, the overhead incurred by dynamic sandboxing (5.4%) is comparable to static sandboxing, as dynamic sandboxing only requires inserting an address masking instruction before memory operations, akin to certain implementations of the static sandboxing approach.

WCET Overhead - Compared to full DFI, OP-DFI achieves a 181% reduction in WCET overhead. WCET overhead is attributed to the unavoidable code switching (0.4%) and sandboxing (2.3%) instrumentation on WCEP. Conversely, the code instrumented for data-flow checking does not add to the WCET of the original task due to the design of OP-DFI: it only checks when slack is available. Code switching typically induces less WCET overhead than sandboxing. This disparity arises because the cost associated with constraint evaluation, a phase of the code switching process, is restricted by a predefined threshold (0.3% of WCET). In contrast, sandboxing occurs with memory accesses on WCEP. Upon further investigation of code switching, the overhead of some instances slightly exceeds the predefined threshold due to the added complexity of context tracking. However, in Autoware and OpenManipulator, the overhead stays below this threshold. This can be ascribed to two factors: The selected constraints can depend directly on inputs, lowering evaluation costs, and some constraints undergo partial evaluation due to missing runtime contexts on the WCEP.

Memory Overhead: Figure 5 shows the breakdown of memory overhead for each system component. Built on top of baseline DFI, OP-DFI shares a common memory overhead (approximately 50%) with baseline DFI due to the utilization of read-write tables for taint tracking. Moreover, OP-DFI introduces additional memory overhead ranging from 19.2% to 31.7% due to various factors, including code ver-

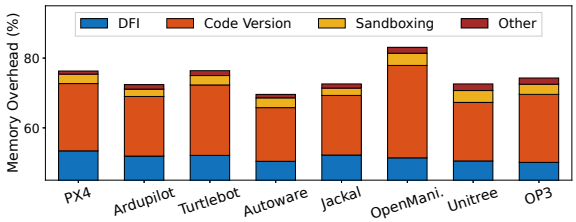


Figure 5: Memory overhead breakdown.

Table 3: Context tracking for slack estimation accuracy

CPS	D. P.	No Ctx	Partial Ctx	Full Ctx
PX4	w	69%	78%	92%
	w.o	53%	65%	
Turtlebot	w	79%	85%	95%
	w.o	58%	73%	
OpenMani.	w	59%	73%	90%
	w.o	41%	61%	

D.P.: dynamic profiling, w: with, w.o: without

sions, sandboxing, and others (such as the pre-computed table for switching target lookup and the shadow stack for return address protection). Among these factors, code versions contribute the most significant overhead, accounting for 18.9% of the memory overhead. The second contributing factor to memory overhead is the sandboxing implemented using MTE, introducing an overhead of 2.7%. This arises because MTE supports isolation only at a 16-byte granularity, necessitating data alignment to accommodate these 16-byte granularity restrictions when adjacent data has different privileges.

Protection Coverage: With a WCET overhead of 2.7%, OP-DFI effectively performs DFI checking on approximately 95.5% of all memory operations and 99.3% of those related to control variables, where the discrepancy arises from the prioritization of control variables during the policy generation phase. Typically, CPS with ample slack (which allows for more resource allocation) and a lower average runtime overhead for DFI operations (thereby consuming fewer computational resources for checks) yield higher coverage scores. Moreover, a greater data and control dependency within a program can reduce the protection coverage. This happens because interdependent data needs to be consolidated into a single set when choosing instrumentation points. As the size of these data sets increases, the granularity of slack utilization becomes coarser, which compromises the protection coverage. For instance, even though OpenManipulator has less available slack (with its average runtime accounting for 21% of WCET) and a higher average DFI runtime cost (120%) compared to Jackal, it achieves better protection coverage. This is attributed to fewer dependencies among its program variables.

8.2 Analysis of Slack Estimation

There are two key factors influencing slack estimation: different levels of context tracking and variations that cannot be captured by inputs a priori.

Table 4: Slack variation

CPS	Mission	Sla. Var.	Est. Acc.	Est. Acc.↑
PX4	takeoff	0.17	77%	2.8%
	flyto	0.19	70%	4.1%
	land	0.25	59%	5.6%
Turtlebot	w.o obstacle	0.08	86%	0.4%
	w obstacle	0.11	72%	3.2%
OpenMani.	grasp	0.27	62%	1.8%
	place	0.29	55%	2.8%
	rotate	0.15	73%	2.2%

Different Levels of Context Tracking: To investigate the impact of varying context levels on slack estimation accuracy, we experimented with three distinct settings: input-only (no context), partial context, and full context. In the input-only setting, the considered inputs are the navigation mode, user command, and position status for PX4; laser data for Turtlebot; and user command, trajectory status, and moving status for OpenManipulator. The partial context represents the program's default configuration, where only a subset of runtime variable updates are automatically selected through optimization. In contrast, the full context involves tracking every branch target encountered during program execution. Table 3 presents the results for the three CPS platforms.

The results indicate that incorporating more context improves the precision of slack estimation. Specifically, when using full context, the accuracy reaches 92%. The residual discrepancy is primarily attributed to inter-process interference. Employing partial context provides an average accuracy increase of 9.6% compared to input-only estimation. Remarkably, for OpenManipulator, the improvement is even more pronounced at 14%. This is because certain loops in its kinematics solver are resource-intensive and heavily dependent on non-input global variables. We also evaluated the effectiveness of dynamic profiling in constraint selection for both input-only and partial context scenarios. The results are indicated in the second column D.P. in Table 3. We observed an average improvement of 15.3% in slack estimation accuracy, underscoring the efficacy of dynamic profiling.

Slack Variation: We further measured slack variation across eight different missions, calculating the normalized difference between the actual slack and the lowest observed slack for the same input. The results, presented in Table 4, show a variation ranging from 0.08 to 0.29, with an average of 0.18. Greater slack variations generally result in reduced slack estimation accuracy because the worst case is assumed for uncertain estimates. To improve slack estimation, one viable approach is to increase the number of path constraints for a more precise execution state estimation, albeit potentially at the expense of performance. By increasing the number of path constraints from the original setting of 16 to 24, we achieved a 2.8% improvement in estimation accuracy.

8.3 Analysis of Code Version

We first evaluated the general protection coverage and runtime performance across different code versions. Then we exam-

Table 5: Properties of different code versions

CPS	Code Version	Ave. Run. Overhead	Cov. Score	Cov. Diff.	Switch Cond.
PX4	highest	118%	1	0.15	>67%
	others	75%	0.83		9%∼67%
	lowest	19%	0.16		<9%
Turtlebot	highest	162%	1	0.21	>75%
	others	116%	0.77		13%∼75%
	lowest	7%	0.09		<13%
OpenMani.	highest	124%	1	0.09	>72%
	others	86%	0.89		16%∼72%
	lowest	5%	0.12		<16%

ined code switching frequency in different missions. Lastly, we manually introduce different numbers of code versions to further explore the trade-offs in selecting such a number during optimization. For the sake of simplifying the discussion, in the rest of this section, we focus on the code versions with the *highest* and *lowest* coverage scores, while grouping all other versions under the category *others*.

Protection Coverage and Overhead: Table 5 presents these two properties as well as the average difference in coverage among different code versions. It is observed that code versions with higher protection coverage generally incur a greater runtime cost due to the increased number of memory operations that need to be checked. This leads to the associated switching condition requiring a larger slack to accommodate the increased runtime cost. Coverage differences among the various code versions are relatively small and even, ranging from 0.09 to 0.21. Notably, even though the code version with the lowest coverage is designed for scenarios with no slack, it can still have a non-zero coverage score (averaging around 0.12 in our evaluation). As long as the memory operations checked in a given code version are not part of the WCEP, these checked data operations will not affect the WCET.

Code Switching: The last column in Table 5 indicates the range of available slack required to switch to a specific code version at the first switch. Among the tested CPSs, Turtlebot has the most stringent switching conditions, requiring more slack due to its higher runtime cost of DFI compared to the other two CPSs. For example, the *highest* code version in Turtlebot requires more than 75% of the WCET as slack to initiate the switch. As for the actual switching frequency, Figure 6 presents both the frequency at which the program switches to a specific version at the first code-switching point

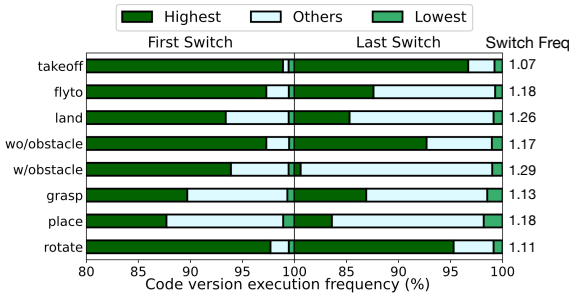


Figure 6: Code switching frequency.

and the frequency at which the program remains on that version after the last switching point. The results indicate that OP-DFI implements proactive protection by maintaining an overly large initial protection region to take advantage of the potential increase in slack while maintaining security coherence. This is illustrated in Figure 6, where the dark green bars in the left subfigure are longer than those in the right subfigure. Moreover, switching primarily occurs at the program's entry point, with the switching frequency in subsequent executions remaining below 0.18, due to the availability of sufficient slack. In approximately 91% of cases, the program eventually switches to the version with the highest coverage for the same reason. Additionally, different missions can result in varying switching frequencies. For instance, the "w obstacle" mission exhibits a higher switching frequency compared to the "w.o obstacle" mission of the same CPS. This difference is due to the presence of randomly occurring obstacles during the mission leading to fluctuations in slack, which cannot be inferred at the program's entry point. Consequently, the updated slack prompts OP-DFI to narrow the protection coverage.

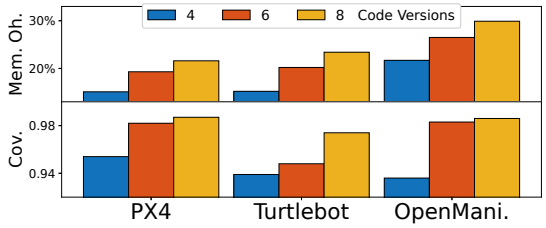


Figure 7: Memory overhead and protection coverage under varying numbers of code versions.

Number of Code Versions: We also investigated the impact of varying the number of code versions. Illustrated in Figure 7, when the number of code versions increases from 4 to 8, the protection coverage experiences a rise of 0.04. However, the memory overhead also witnesses a 7.6% increase. Our results indicate that an increase in the number of code versions leads to improved protection coverage, effectively creating a more fine-grained security policy. It's also noteworthy that the growth in memory overhead is not linear with the number of code versions. This is because different code versions can share the common code pages, and OP-DFI leverages the MMU to map the common physical code pages to multiple code versions, thereby reducing memory overhead.

8.4 Adaptive Attack Analysis

Analysis Principle: While the level of protection provided by OP-DFI is bounded by the available slack, the need to exhaust slack also constrains the attackers on the adversarial input, since the input must satisfy specific path constraints to cause a significant expansion in the task run-time. To gain a more holistic understanding of the practical level of protection for CPS systems, this trade-off is analyzed through the lens of

Table 6: Constraint on control flow and input value

CPS	Est. Slack	Constraint		Cov. Score
		control flow	input value	
PX4	0	27%	0.042%	0.16
	0∼20%	42%	0.73%	0.43
	0∼40%	71%	8.2%	0.75
	0∼80%	87%	15.3%	0.95
Turtlebot	0	24%	0.007%	0.09
	0∼20%	47%	1.6%	0.36
	0∼40%	67%	5.5%	0.73
	0∼80%	84%	14.4%	0.84
OpenMani.	0	22%	0.015%	0.12
	0∼20%	38%	0.063%	0.41
	0∼40%	56%	4.8%	0.71
	0∼80%	90%	27.5%	0.91

the exploitability by an adaptive adversary in this section. Even though the concrete techniques to implement a shellcode could differ from one stance to another, there are often two necessary conditions to successfully exploit a vulnerability in the system, namely, the ability to reach vulnerable functions with attackers' input and the ability to craft the input to trigger the vulnerability upon delivery. Note that even if the input is supplied by the attacker, there can be various logic in the program, such as data sanitization, preventing arbitrary manipulation on such input.

Analysis Approach: The first exploitability metric is adversarial access to the security-sensitive functions, since function calls, like unsafe C library function calls and system calls [11, 20, 23], are frequently used for exploitation. However, not all the function calls matter, only those with adversarial inputs flowing into can be exploited. Thus, only calls to security-sensitive function with adversarial information flows are measured. To provide a quantitative measurement, it is quantified as the percentage of reachable security-sensitive locations, which are defined as calls to security-sensitive functions. The second exploitability metric is the adversary's ability to manipulate the input. To obtain the ranges of possible manipulation, the path constraints from the worst-case execution path are converted to a logic formula that limits the range. To obtain a quantitative measurement, the percentage of the modifiable range over the full range of the input is measured. It is important to note that the measurement for exploitability requires considerations of many subtle details and remains an open research challenge [52]. The two basic metrics measured in this study are merely the first step towards quantitatively understanding the exploitability in the context of OP-DFI, and in the future, we intend to further develop this research.

Analysis Results: The quantitative measurement of the constraints at different slack levels are shown in Table 6. As the results show, in the worst-case scenario where there is no slack, the reachable security-sensitive locations are constrained to 24.3%, and only 0.02% of the program's input value range can be utilized for generating exploits. This limitation arises because, in order to traverse longer paths, the input must satisfy all constraints along these paths, thereby restricting control flow and input value. As the slack level increases, attackers gain greater flexibility to craft exploit payloads and access more security-sensitive locations. However, concurrently, the protection offered by OP-DFI also increases due to the augmented amount of slack (with a coverage score of 0.90 when slack is 80% of WCET), providing enhanced defense. A concrete example can be found in Appendix C.2.

9 Security Analysis

Security of System Implementation. First, attackers may attempt to jump around the instrumented reference monitor to bypass security checking. However, OP-DFI leverages the hardware feature ARM BTI to efficiently enforce a coarse-grained CFI, which ensures the attacker can only jump to the function entries and return to callsites. As a result, all data operations are checked, and the control transfer between code versions can only occur in code switching units given that all indirect jumps are masked into an address range. Second, OP-DFI uses MTE to securely maintain the write protection of data. All data created from the protected region is associated with specific MTE tags, and only the code within the protected region carries these same MTE tags. Any write operations from the unprotected region will alter the original tags of the protected data, which will subsequently be detected when the data is read by the protected region. At runtime, OP-DFI may dynamically reduce the scope of the protected region via code version switches. In the newly activated code version, the MTE tags in the removed region are hardcoded to be different from those in the protected region. Consequently, after such a code version switch, any attempt from the removed region to directly write on the protected data will also be detected. Furthermore, since the removed region has no dependency with the remaining protected region, the sandboxing effectively prevents data corruption from the unprotected region. Lastly, the metadata for OP-DFI and the software-based DFI enforced in the protected region are also protected by ARM MTE. OP-DFI ensures that all MTE tags are hardcoded and reserve specific tags to the metadata, ensuring that only designated RMs can modify it.

Security of Data-flow Protection. Although OP-DFI does not guarantee complete protection for all data flows, its adaptability allows it to optimally secure a desired subset of data. For instance, in the eight platforms we evaluated, OP-DFI can be customized to achieve an average protection coverage of 99.3% for control variables. Furthermore, for variables that are expected to be protected, OP-DFI offers comparable security guarantees as full DFI. Since OP-DFI enforces DFI to check data flows in the protected region, any illegal data flows within the protected region that deviate from DFG can be detected. An attacker may attempt to modify unprotected data to influence the protected data. However, this is not possible because OP-DFI maintains security coherence, taking into account both the dependency and the protection duration of

the protected region. The former ensures that at any given time, the protected region encompasses all data and control dependencies of the protected data. Additionally, the latter ensures that the shrunken protected region, by the end of the task, remains protected throughout the entire iteration period.

Real-time Guarantee. The violation of real-time guarantees can arise from either the overestimation of slack or when the actual enforced DFI exceeds the capability of the estimated slack. For the first case, our slack estimation provides a lower bound for the actual slack, as we proved in Claim 1. Regarding the second case, the DFI enforcement policy is generated under the strict constraints of the slack budget, as outlined in Appendix A.3. Consequently, its actual execution time cannot exceed the capability allotted by the given slack.

10 Related Work

Information Flow Integrity: Among the existing work on DFI, RT-DFI [13] optimizes DFI tag checks to reduce the overhead for real-time systems. KENALI [50] supports partial DFI to only protect selected data. TMDFI [40], HDFI [51], and Trustflow [14] reduce the runtime overhead of DFI but require customized hardware. OP-DFI complements existing work by utilizing slack to minimize WCET overhead. Furthermore, some existing works on CFI take real-time into consideration [21, 47, 58, 65]. However, most of them focus on average runtime efficiency, while OP-DFI focuses on minimizing WCET overhead using slack.

Opportunistic Execution: Opportunistic execution [25, 35, 41] leverages slack to perform additional workloads. For example, [25, 35] utilize slack to execute extra inter-process real-time tasks, which can include security checks [25]. However, [25] is limited to examining program static properties rather than runtime behaviors. On the other hand, [41] employs slack measured by the timer for intra-process online monitoring, albeit requiring customized hardware. OP-DFI complements these works by using slack to assess program runtime behavior without the need for customized hardware.

Parametric WCET: The parametric WCET analysis [7, 8, 12, 15, 28, 43, 56] aims to derive a function that expresses the WCET of a program in terms of its input parameters, software-related factors (e.g., flow constraints), and hardware-related factors (e.g., timing impacts of cache and architecture). However, existing works manly focus on offline analysis.

11 Discussion and Limitation

Extending Optimization Across Multiple Execution Iterations: The current design of OP-DFI focuses on maximizing protection on a single job release. However, for periodical tasks in CPS, there are usually strong temporal dependencies among different real-time tasks or different jobs of the same task. Though such dependency has not been an issue for platforms in our evaluation, it is entirely possible that a large amount of control variables have to be protected in the beginning when slack is not available. A potential mitigation is to use sanitizers to bring the critical control variable back to protection or reset it to the default state, leveraging the physical momentum to mitigate the control impact.

Dependence on Hardware Feature: While OP-DFI utilizes existing hardware features ARM MTE and BTI for runtime enforcement, it is also possible to use software-based fault isolation and software-based CFI to realize the enforcement.

Memory Overhead: The current implementation of OP-DFI incurs a non-negligible memory overhead of 23.1% over that of the full DFI. This increase is largely attributed to the utilization of multiple code versions designed to facilitate changes in security policy enforcement. Although memory usage is already optimized by sharing code pages common to different versions, as the number of code versions increases, the effectiveness of this optimization can diminish because common code regions become more segmented, resulting in fewer 4KB (page-size) common code regions. Several potential mitigation solutions exist. First, more common code pages can be generated by adjusting the code layout during compilation. Second, finer-grained code region sharing can be achieved by decomposing the execution of a code version into the executions of multiple smaller code regions. Then, the sharing of common regions can be realized using trampolines that transit execution among different regions. Lastly, it is possible to trade space expansion with computational overhead by dynamically updating the policy in place.

12 Conclusion

In this paper, we propose a security primitive, opportunistic DFI, to opportunistically perform in-line data-flow checking using the opportunity of slack in real-time systems. To enforce opportunistic DFI, a system OP-DFI is proposed, for which an input/context-dependent slack is constructed, the real-time property and security coherence are modeled, and dynamic runtime enforcement mechanisms are designed. A prototype is implemented on the AArch64 platform and evaluated on eight real-time CPSs.

Acknowledgment

We thank the reviewers for their valuable feedback. This work was partially supported by the NSF (CNS-2141256, CPS-2229290, CNS-1916926, CNS-2038995, CNS-2154930, CNS-2238635), ARO (W911NF2010141), and Intel.

References

[1] Arm mte. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[2] Linux kernel mte support. https://lwn.net/Articles/834289/.

[3] Mercedes-benz mbux security research report. https://keenlab.tencent.com/en/whitepapers/Mercedes_Benz_Security_Research_Report_Final.pdf.

[4] New exploitation techniques and defenses for dop attacks. https://techxplore.com/news/2019-03-exploitation-techniques-defenses-dop.html.

[5] Arm architecture reference manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/fc, 2022.

[6] Timeweaver. https://www.absint.com/timeweaver/index.htm, 2023.

[7] Ernst Althaus et al. Precise and efficient parametric path analysis. *ACM SIGPLAN Notices*, 2011.

[8] Sebastian Altmeyer et al. Parametric timing analysis for complex architectures. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[9] Ardupilot project. https://ardupilot.org/.

[10] Karl Johan Åström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2021.

[11] Thanassis Avgerinos et al. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, 2011.

[12] Clément Ballabriga et al. Context-sensitive parametric wcet analysis. In *WCET*, 2015.

[13] Nicolas Bellec and et al. Rt-dfi: Optimizing data-flow integrity for real-time systems. In *Euromicro Conference on Real-Time Systems*, 2022.

[14] Cyril Bresch and et al. Trustflow: A trusted memory support for data flow integrity. In *Computer Society Annual Symposium on VLSI*, 2019.

[15] Stefan Bygde et al. An efficient algorithm for parametric wcet calculation. *Journal of Systems Architecture*, 2011.

[16] Cristian Cadar et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, 2008.

[17] Nicholas Carlini et al. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security*, 2015.

[18] Miguel Castro et al. Securing software by enforcing data-flow integrity. In *Operating Systems Design and Implementation*, 2006.

[19] Shuo Chen et al. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005.

[20] Lucas Davi et al. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[21] Yufei Du et al. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *USENIX Security*, 2022.

[22] Mahmoud A Elmohr et al. Em fault injection on arm and risc-v. In *International Symposium on Quality Electronic Design*. IEEE, 2020.

[23] Seyedhamed Ghavamnia et al. Temporal system call specialization for attack surface reduction. In *USENIX Security*, 2020.

[24] Jan Gustafsson et al. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *IEEE Real-Time Systems Symposium*, 2006.

[25] Monowar Hasan et al. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *2016 IEEE Real-Time Systems Symposium*, 2016.

[26] Hong Hu et al. Automatic generation of data-oriented exploits. In *USENIX Security*, 2015.

[27] Hong Hu et al. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*, 2016.

[28] Benedikt Huber et al. A formal framework for precise parametric wcet formulas. In *12th International Workshop on Worst-Case Execution Time Analysis*, 2012.

[29] Kyriakos K Ispoglou et al. Block oriented programming: Automating data-only attacks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[30] Jackal. https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/.

[31] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 1986.

[32] Daniel Kästner et al. Timeweaver: A tool for hybrid worst-case execution time analysis. In *19th International Workshop on Worst-Case Execution Time Analysis*, 2019.

[33] Shinpei Kato et al. Autoware on board: Enabling autonomous vehicles with embedded systems. In *ACM/IEEE 9th International Conference on Cyber-Physical Systems*, 2018.

[34] Mustakimur Rahman Khandaker et al. Origin-sensitive control flow integrity. In *USENIX Security*, 2019.

[35] Angeliki Kritikakou et al. Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems. In *International Conference on Real-Time Networks and Systems*, 2014.

[36] Angeliki Kritikakou et al. Run-time control to increase task parallelism in mixed-critical systems. In *26th Euromicro Conference on Real-Time Systems*, 2014.

[37] Ao Li et al. From timing variations to performance degradation: Understanding and mitigating the impact of software execution timing in slam. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2022.

[38] Ao Li et al. Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference. In *IEEE Real-Time Systems Symposium*, 2022.

[39] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. 1973.

[40] Tong Liu et al. Tmdfi: Tagged memory assisted for fine-grained data-flow integrity towards embedded systems against software exploitation. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering*, 2018.

[41] Daniel Lo et al. Slack-aware opportunistic monitoring for real-time systems. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, 2014.

[42] Paul Lokuciejewski and Peter Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In *21st Euromicro Conference on Real-Time Systems*, 2009.

[43] Amine Marref. Evolutionary techniques for parametric wcet analysis. In *12th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[44] Derrick McKee et al. Preventing kernel hacks with hakc. In *Proceedings Network and Distributed System Security Symposium*, 2022.

[45] Lorenz Meier et al. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *ICRA*. IEEE, 2015.

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

[47] Thomas Nyman et al. Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium*, 2017.

[48] Op3. https://emanual.robotis.com/docs/en/platform/op3/introduction/.

[49] David Sehr et al. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security*, 2010.

[50] Chengyu Song et al. Enforcing kernel security invariants with data flow integrity. In *Network and Distributed System Security Symposium*, 2016.

[51] Chengyu Song et al. Hdfi: Hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy*, 2016.

[52] Octavian Suciu et al. Expected exploitability: Predicting the development of functional vulnerability exploits. In *USENIX Security*, 2022.

[53] Turtlebot. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[54] Unitree robotics. https://github.com/unitreerobotics.

[55] Victor Van der Veen et al. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[56] Emilio Vivancos et al. Parametric timing analysis. In *Proceedings of the ACM SIGPLAN workshop on optimization of middleware and distributed systems*, 2001.

[57] Robert Wahbe et al. Efficient software-based fault isolation. In *Proceedings of the 14th ACM symposium on Operating systems principles*, 1993.

[58] Robert J Walls et al. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems*, 2019.

[59] Jinwen Wang et al. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *Symposium on Security and Privacy*. IEEE, 2022.

[60] Jinwen Wang et al. Ari: Attestation of real-time mission execution integrity. In *USENIX Security*, 2023.

[61] Jinwen Wang et al. Secure and timely gpu execution in cyber-physical systems. In *ACM Conference on Computer and Communications Security*, 2023.

[62] Hang Wu et al. Pid controllers: Design and tuning methods. In *IEEE Conference on industrial electronics and applications*, 2014.

[63] Zhiyuan Yu et al. Security and privacy in the emerging cyber-physical world: A survey. *Communications Surveys & Tutorials*, 2021.

[64] Ning Zhang et al. Truspy: Cache side-channel information leakage from the secure world on arm devices. *Cryptology ePrint Archive*, 2016.

[65] Jie Zhou et al. Silhouette: Efficient protected shadow stacks for embedded systems. In *USENIX Security*, 2020.

A Additional Design Details

A.1 Proof of Parametric Slack Estimation

CLAIM 1 Given the constraint evaluation results S_Ω for a set of selected constraints Ω_s, Algorithm 1 provides an under-approximated worst-case slack estimate $\mathcal{B}$ of the program.

Based on the definition of slack: slack = WCET - execution time, we define $\mathcal{W} = WCET - \mathcal{B}$. We then show the equivalence of Claim 1: Algorithm 1 provides an over-approximated worst-case time estimate $\mathcal{W}$ of the program, given the constraint evaluation results S_Ω for a set of selected constraints Ω_s.

Proof. Let $\mathcal{P}$ be the program under analysis, $\mathcal{K}$ be the worst-case execution time analysis tool, $\{p_1, p_2, ..., p_n\}$ be the set of all execution paths for $\mathcal{P}$, $\{t_1, t_2, ..., t_n\}$ be the corresponding (over-approximated) execution time estimates for the paths, and $\{\Omega_1, \Omega_2, ..., \Omega_n\}$ be the corresponding path constraints. Each Ω_i is a constraint statement consisting of the concatenation of the individual branch constraint $\{\omega_i^1, \omega_i^2, ..., \omega_i^j\}$. As previously defined in Algorithm 1, S_Ω represents the results of evaluating the constraints in the set Ω_s, where $\forall \omega_s \in \Omega_s$, $\omega_s \in \{\Omega_1 \cup \Omega_2 \cup \ldots \cup \Omega_n\}$.

We make the assumption that given the program $\mathcal{P}$ and the worst-case analysis tool $\mathcal{K}$, $\mathcal{K}(\mathcal{P}) \rightarrow \{(p_1, t_1), ...(p_n, t_n)\}$, where t_i is the over-approximated estimate of the execution time along p_i. We also assume there exists a logical solver Ψ that can determine whether a path p is feasible or not.

To obtain an over-approximated worst-case time estimate, $\mathcal{K}$ can simply output t_{max}, where $t_{max} = max\{t_1, t_2, ..., t_n\}$. To obtain a tighter bound, some WCET tools also eliminate infeasible paths, therefore the output becomes $t_{max} = max\{t_1, t_2, ..., t_m\}$ where $t_i \in \{t_1, t_2, ..., t_n\}$, $\Psi(p_i) \rightarrow true$. Intuitively, Algorithm 1 goes one step further and eliminates the infeasible paths using the known states in S_Ω. Mathematically, the output is now $t_{max} = max\{t_1, t_2, ..., t_q\}$ where $\forall t_i \in \{t_1, t_2, ..., t_n\}, \Psi(p_i) \rightarrow true \wedge feasible(p_i, r_s)$ for $\forall (\omega_s : r_s) \in S_\Omega, \omega_s \in \Omega_i$. $feasible(p_i, r_s)$ is true either r_s is *unknown* or r_s does not violate the condition of p_i. As a result, among all feasible paths, the algorithm still selects the longest running time, and since this running time estimate is over-approximated, this algorithm still returns an over-approximation of the worst-case execution time.

We can prove this by contradiction. Let t_w be the worst-case time output by Algorithm 1 given $\mathcal{P}$ and S_Ω, with the corresponding worst-case execution path as p_w. Since this is the worst-case execution time, that means for all feasible paths, $\mathcal{F}$, where $\Psi(p_i) \rightarrow true \wedge feasible(p_i, r_s)$ for $\forall (\omega_s : r_s) \in S_\Omega, \omega_s \in \Omega_i$ given the known constraint evaluation results S_Ω, t_w is the largest number in the set, i.e. $\forall t_i \in \mathcal{F}, t_w \geq t_i$. Let p_z be the program execution path that leads to running time estimation of t_z, however $t_z > t_w$. This creates a contradiction between two complementary conditions. If $p_z \in \mathcal{F}$, then it contradicts with the max selector statement where $\forall t_i \in \mathcal{F}, t_w \geq t_i$. If $p_z \notin \mathcal{F}$, then it contradicts either the constraint evaluation results S_Ω or the original path feasibility $\Psi(p_z) \rightarrow true$.

A.2 Quantifying Runtime Cost

To quantify the runtime cost of a specific path, the path is divided into subpaths based on slack estimation checkpoints along its course. However, due to the uncertainty of the estimated execution state related to other paths, we enumerate the complete set of potential intermediate estimated execution states for each subpath. This set is denoted as $\mathcal{S}^*(p, S_\Omega) = \{\langle S^{p_{[0]}}, S^{p_{[1]}}, \ldots \rangle\}$, where $S^{p_{[i]}}$ represents a potential intermediate estimated execution state on subpath $p_{[i]}$. Subsequently, the runtime cost for a path given an estimated execution state, denoted as $\Delta(p, S_\Omega|\pi)$, is established as the highest runtime cost among different potential intermediate execution states:

$$\Delta(p, S_\Omega|\pi) = \max_{\langle S^{p_{[0]}}, S^{p_{[1]}}, \ldots \rangle \in \mathcal{S}^*(p, S_\Omega)} \sum_{i}^{n} \delta(\pi(S^{p_{[i]}}), p_{[i]}) \quad (3)$$

where $\pi(S^{p_{[i]}})$ represents the set of checked memory operations selected by the security policy with the intermedi-

ate estimated execution state of subpath $p_{[i]}$. Furthermore, $\delta(\pi(S^{p_{[i]}}), p_{[i]})$ is the runtime cost on subpath $p_{[i]}$ arising from the chosen checked memory operations $\pi(S^{p_{[i]}})$, and it can be computed based on instruction count. Finally, the runtime cost under an estimated execution state S_Ω can be obtained as the largest runtime cost among all paths corresponding to that estimated state: $\Delta(S_\Omega, \pi) = \max_{p \in \mathrm{P}} \Delta(p, S_\Omega | \pi)$.

A.3 Policy Optimization

Optimization Problem Formulation: According to the definition of a security policy π, it can be parameterized by the checked memory operations across different code versions $\Phi = \{\theta_0, \theta_1, ...\}$, as well as the switching rules utilized at different slack estimation checkpoints $\{\pi_0, \pi_1, ...\}$. The set of selected constraints can be denoted as $\Omega_s = \{\omega_0, \omega_1, ...\}$. The goal of the optimization is to find the optimal parameters that satisfy the specified constraints:

$$
\begin{aligned}
(\Omega_s, \pi)^* = & \arg\max_{\Omega_s, \pi} \mathcal{C}(\Omega_s, \pi) \\
\text{s.t.} \quad & \Delta_\Omega(\Omega_s) \leq \varepsilon_\Omega \\
& \Delta(S_\Omega, \pi) \leq \mathcal{B}(S_\Omega), \text{ for all } S_\Omega \\
& \mathcal{D}(\theta) = \theta, \text{ for all } \theta \in \Phi \\
& \pi(S) \supseteq \pi(S'), \text{ for all } (S, S') \in \mathcal{S}^*. \\
& |\Omega_s| = N_\Omega \\
& |\Theta| = N_\Theta
\end{aligned}
\tag{4}
$$

where $\Delta_\Omega(\Omega_s)$ is the evaluation cost of the constraint formula (which can be obtained through the instruction count of the formula) and is bounded by a threshold ε_Ω; $\mathcal{D}(\theta)$ is the dependency of θ; (S, S') is the estimated execution state before and after the transition, and $\mathcal{S}^*$ is the set of all possible transitions; $|\Omega_s|$ is the number of selected constraints where constraints with the same formula are deemed identical, and it is set to N_Ω; $|\Phi|$ is the number of code versions and is set to N_Φ.

The first constraint bounds the evaluation cost of the constraint formula. The second constraint pertains to the real-time requirement, ensuring that the runtime cost of checked memory operations remains within the available slack. The third and fourth constraints relate to security coherence, ensuring that the code version includes all necessary dependencies and can only shrink following each code switch. The fifth constraint limits the table size to $O(2^{N_\Omega})$ entries, and the sixth constraint restricts the number of code versions.

B Additional Details on Implementation

B.1 Optimization Engine

In order to generate the optimal parameters, the search space is initially reduced to facilitate optimization. Considering that the evaluation cost of selected constraints is bounded, constraints with intricate symbolic formulas can be excluded from the search space. Additionally, by leveraging the security-coherence constraint, the code version must encompass all dependencies, and the switching rule exclusively transitions to versions with fewer checked memory operations, resulting in a further reduction of the search space.

Subsequently, the optimal parameters can be approximately determined through the utilization of Mixed-Integer Linear Programming (MILP) and Genetic Algorithm. To elaborate, the initial optimization parameters of the genetic algorithm are first set. Specifically, since high slack estimation usually provides more opportunities for protection, the initial parameters for selected constraints are set as the set of constraints that can yield the highest average estimated slack. After establishing the initial search state, the genetic algorithm iteratively performs the following steps: it mutates the selected constraints, and after each mutation, MILP is employed to facilitate the search for the corresponding optimal switching rules and code versions. This iterative process continues until convergence is achieved or timeouts.

C Additional Details on Evaluation

C.1 Evaluation Setup

Testing mission used in evaluation: Table 7 lists all the missions employed to test the eight CPS platforms in this paper. These missions encompass trajectories or sequences of actions. For drones and autonomous vehicles, the CPS platforms are tasked with executing circular trajectory missions. The robotic arm (OpenManipulator) is assigned the duty of repeatedly performing actions involving object grasping, rotation, and placement. Similarly, the quadruped robot (Unitree) and humanoid (OP3) are required to cyclically complete tasks such as standing up, walking, turning around, and sitting down. An asterisk (*) in the table denotes that the official repository does not provide specific test missions; instead, it offers simulated scenes. As a result, we manually curated the missions using the provided scenes.

Table 7: Testing missions

	Mission Type	Mission Name
PX4	Trajectory	FW_mission_1.plan
Ardupilot		copter_mission.txt
Turtlebot		turtlebot3_house*
Autoware		nishishinjuku_autoware_map*
Jackal		agriculture_world*
Unitree	Actions	stairs*
OP3		walking demo
OpenMani.		pick and place demo

Instruction Analogs: Given that the prototype utilizes

the hardware features ARM MTE and BTI, to the best of our knowledge, there are no publicly available development boards that support ARM MTE and BTI at the time of writing. Consequently, instruction analogs are employed to assess performance. Instruction analogs are a series of instructions that consume similar CPU cycles and memory footprint but do not perform actual checks [44]. According to the description, considering the negligible overhead of MTE and BTI, we use regular memory operations to simulate MTE-specific memory operations. For example, we analog an MTE-specific store instruction (`STGP`) with a regular store instruction (`STR`). Additionally, we analog a BTI instruction as an exclusive-or (`EOR`) instruction.

WCET Meaurement: We utilize the aiT TimeWeaver [6], a timing analysis tool, for WCET measurement. This tool can take CoreSight ETM trace packets as inputs to obtain architectural-level timing information. Among the four boards, only Jetson Nano (Cortex-A57) and Jetson AGX Orin (Cortex-A78) support Coresight ETM, while we were unable to find the CoreSight ETM supporting documentation for the other two, Raspberry Pi 3 (Cortex-A53) and Raspberry Pi 4 (Cortex-A72). To conduct WCET measurement on the Cortex-A53 and Cortex-A72 cores, we thus utilize the Cortex-A53 and Cortex-A72 cores available on the Juno Development Board, where CoreSight ETM is available. This serves as an alternative method to obtain architectural-level timing information for the Cortex-A53 and Cortex-A72 processors.

C.2 Adaptive Attack Evaluation

An Example of Constraint on Input Value: To justify the reasonableness of our metric — which involves constraints on the input value range — we refer to a real-world example from PX4 for clarification. Listing 2 shows a scenario where the adaptive attacker is unable to simultaneously push the task onto the worst-case execution path and trigger the exploit. The code snippet is taken from the `mavlink_receiver.cpp` file in the PX4 project, with irrelevant code omitted for simplicity.

The function `handle_message_statustext()` (on line 01) is responsible for parsing the input messages regarding system status and then publishing the parsed information as logs if necessary (the condition is on line 05). At the outset of this function, line 03 invokes `memcpy()` to copy the content of `msg` to a local variable `statustext`. Initially, length checking exists before the copy to prevent out-of-bounds writes. For the attack demonstration, we replaced this with an unsafe version on line 04, allowing the attacker to craft a large `msg` to trigger a buffer overflow, a preliminary step in exploiting the system. Concretely, the attacker needs to create an exploit payload that is long enough to overwrite data beyond the buffer size of `statustext`. However, such a crafted exploit payload cannot trigger the WCEP (line 06) and thus cannot lead OP-DFI to predict WCET. The reason is that the WCEP depends on a `true` return value from the function `should_publish_current()`. However, if `statustext` is overwritten, this function will always return `false`. This is because the function `should_publish_current()`, on line 09, inspects the content of `statustext`. If `statustext` contains no `\0`, the task will return `false`, preventing the task from entering the WCEP. In summary, to steer the task onto the WCEP, the input value must remain within the value limited by the size of `statustext`. Adhering to this constraint prevents the triggering of a buffer overflow, not to mention successful exploitation.

```
01  void handle_message_statustext(mavlink_message_t *msg) { /*Attacker's input*/
-02     uint8_t len = msg->len < STATUSTEXT_LEN? msg->len : STATUSTEXT_LEN;
-03     memcpy(statustext, _MAV_PAYLOAD(msg), len);
+04     memcpy(statustext, _MAV_PAYLOAD(msg), msg->len);
05     if (_mavlink_statustext_handler.should_publish_current(statustext)) {
06         _log_message_pub.publish(_mavlink_statustext.log_message()); /* WCET Path */
       }
   }

07  bool should_publish_current(const mavlink_statustext_t &msg_statustext,...) {
08     /* . . . */
09     const bool found_zero_termination =
strnlen(msg_statustext.text, sizeof(msg_statustext.text)) < sizeof(msg_statustext.text);
       /* False for oversize input */
10     if (found_zero_termination) {
11       return true; /* Need to be true for worst-case execution path */
12     } else {
13         return false;}}
```

Listing 2: Example of constraint on input value.

In general, the constraints imposed by the WCEP always restrict the search space during payload generation. Hence, we leverage these restrictions to quantify the trade-off an attacker faces when attempting to both trigger the WCEP and launch an exploit.

C.3 Comparison to CFI

Table 8: Comparison to CFI

System	Information Flow Checking		Runtime Overhead	
	data flow	control flow	average	WCET
RECFISH [58]	0	100%	25%, 30%	-
Kage [21]	0	100%	6.1%	5.1%
CFI CaRE [47]	0	100%	13%∼513%	-
Silhouette [65]	0	100%	3.1%	2.8%
OP-DFI	96.6%	100%	137%	2.6%

Table 8 contrasts OP-DFI with CFI works tailored to real-time systems [21, 47, 58, 65]. We evaluated [21, 65] on STM32L475 and STM32F469 respectively, while using the reported numbers for others due to lack of source code access. The results indicate that OP-DFI incurs a higher average runtime overhead (135%) due to its comprehensive protection for both control and data flow (96.6% of memory operations and 100% of control transfers are checked) for stronger protection. Regarding the WCET overhead, which is more critical as it directly affects the hardware requirements, OP-DFI (2.6%) is comparable to that of the listed CFIs (3.9%).