Partial Context-Sensitive Pointer Integrity for Real-time Embedded Systems

Yujie Wang[§*], Cailani Lemieux-Mack[†*], Thidapat Chantem[‡], Sanjoy Baruah[§], Ning Zhang[§], Bryan C. Ward[†]
[§] Washington University in St. Louis, [†] Vanderbilt University, [‡] Virginia Tech

2024 IEEE Real-Time Systems Symposium (RTSS) | 979-8-3315-4026-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/RTSS62706.2024.00042

***Abstract*—Safety- and mission-critical cyber-physical systems (CPSs) require temporal correctness to ensure safe physical behavior. This manifests as strict timing requirements, which cannot be missed at runtime. Counter-intuitively, this implies that real-time tasks can be delayed so long as they remain guaranteed to meet their deadlines. This paper explores how extra time in a schedule can be analytically recapitalized for the purpose of applying stronger security protection within individual tasks at compile time. This is achieved through the development of a *partial* context-sensitive pointer-integrity framework (ParCSPI). In this framework, more fine-grained policies can be enforced, with greater runtime overheads, where so doing does not violate real-time constraints. A whole-system optimization framework based upon a mixed-integer linear programming approach to fixed-priority response-time analysis is used to identify precisely which contexts can be checked within the available system-wide time while maximizing system-wide security.**

ParCSPI leverages Arm pointer authentication (PA) to encode context-based equivalence classes into the modifiers of the pointer signature and is implemented using a customized program analyzer and LLVM compiler passes. An evaluation of ParCSPI is presented that includes per-task and system-wide overhead and security tradeoffs, as well as a demonstration on a real-world CPS. Empirical results are presented showing that ParCSPI achieves up to 62% pointer-integrity protection with only 10% worst-case execution time (WCET) overhead, and can find optimal security trade-offs in complex real-time task sets as well as approximate them in reasonable time.

I. Introduction

With the recent advances in autonomous systems, cyber-physical systems (CPSs) are playing increasingly important roles in supporting the daily necessities of our society [1]. Despite their lack of memory safety, C, C++ and other memory-unsafe languages are still pervasive in modern software, especially in embedded and real-time systems. Given the ubiquity and importance of many CPSs, their security is paramount. While enhanced security measures are highly desirable, there exists a well-known dilemma concerning the trade-off between security and performance. This paper addresses the fundamental question of how to maximize security protection without compromising the temporal correctness of the system.

The particular defensive approach we consider is *pointer integrity*. To understand the protection afforded by pointer integrity, we must first consider the mechanics of memory-corruption-based attacks. Attackers have demonstrated how memory-corruption vulnerabilities, such as buffer overflows, can be exploited to corrupt code pointers to redirect control flow in a program to a malicious target, or to corrupt data pointers to modify critical memory content [2], [3], [4], [5], [6], [7]. Pointer integrity seeks to mitigate these types of threats by ensuring that the pointers that are traversed at runtime have not been maliciously corrupted, thereby not diverting control flow to an attacker-controlled target.

One such recent approach to ensuring pointer integrity is through the use of *pointer authentication* (*PA*). Modern Arm processors have support for a technology called Pointer Authentication, which is a hardware-assisted security capability to verify that a pointer has not been corrupted. Arm PA has been used in prior work to realize pointer integrity to thwart large classes of memory-corruption vulnerabilities [8], [9], [10], [11] while keeping the performance overhead low.

The PA works by using the pointer value and a modifier, along with a system-maintained key, to generate a *pointer authentication code (PAC)* as a signature for authenticating access to the pointer. Upon memory access, built-in hardware-based pointer-authentication checks enforce various pointer-security policies. It is common practice to encode specific security contexts using the modifier to further customize the granularity of the security policy [8], [9], [10].

The direct application of PA cannot defend against more advanced pointer-reuse attacks. In such attacks, the adversary overwrites one pointer with another pointer value and its corresponding modifier and PAC. In this case, the attacker does not need to reverse engineer the PAC; instead, they can copy it along with another valid pointer value. However, this limits the attacker in terms of the valid pointer targets they can encode in a malicious pointer to only those pointers that are valid at that point, given the adopted policy. More formally, there exists an *equivalence class* (*EC*) of possible valid pointer targets [8], [10]. Inspired by context-sensitive information-flow analysis [12], [13], [14], one possible solution to reduce the size of the ECs and protect both code and data pointers is to leverage *context sensitivity*. For example, a pointer may be valid at a particular line of code, but it is only valid for one specific sequence of function invocations that lead to that line of code, not for all sequences. In other words, it is only valid in one calling context.

Unique Challenges for Context Sensitivity in Real-time Cyber-physical Systems: While context sensitivity is a powerful mechanism to significantly improve system security, context maintenance often imposes prohibitive run-time overheads. This is because maintaining context requires track-

*Equal Contribution.

979-8-3315-4026-5/24/$31.00 ©2024 IEEE
DOI 10.1109/RTSS62706.2024.00042

ing this information dynamically at runtime. Furthermore, checking this context information can also incur significant runtime overhead. While full context-sensitive pointer integrity provides the strongest security, it may not be practical given its high runtime overhead. Indeed, in preliminary experiments, we found worst-case overheads of 273% in ArduPilot [15] when applying full context sensitivity (See §III for additional details). This paper addresses the question of which context is worth preserving given a limited amount of allowable overhead.

ParCSPI: Partial Context-Sensitive Pointer Integrity. Building on the observation that full context sensitivity imposes prohibitive runtime overhead on real-time systems, we propose to only leverage *partial context sensitivity for pointer integrity* (ParCSPI). Context-sensitive pointer integrity can be relaxed along two dimensions, the granularity of context and coverage of protection. Our goal is to maximize the protection possible for a given level of inflation of the WCET. From the context perspective, the selection of most effective subset of contexts to maintain and leverage to improve security is challenging. Some context maintenance has no additional cost, if it is only maintained and checked on a control-flow path that is not the worst case – such checks should be applied wherever possible, while others along the worst-case execution path (WCEP) do affect the WCET. From the coverage perspective, we've found through analyzing the ArduPilot source code, many of the paths are never executed in certain configurations and environments. Adding protection to them will expand the WCET, but provide limited security benefits in the intended operating environment. Based on this observation, ParCSPI supports incorporating the expected input distribution (cyber and physical inputs) into the quantification of security protection, providing a more realistic threat model for the intended environment. Through this exercise, protection for certain regions of the application is reduced or removed because of low probability of execution (such as extreme physical conditions) or zero probability of execution (such as different platform configurations). Ultimately, our analysis profiles these execution paths and considers which context to maintain, as well as the size of the EC reduction (and hence security) afforded by different degrees of context sensitivity.

To offer holistic protection at the system level, it important to temporally tailor context sensitivity not just to a single task, but to the system as a whole. Not all tasks have the same level of security criticality (e.g., some tasks may not process remote inputs or control safety-critical devices). Furthermore, providing stronger protection to certain tasks may allow for reduced protection in others. To address this problem, we build upon our per-task results and develop an optimization approach based on Mixed Integer Linear Programming (MILP) response-time analysis. This approach optimizes the per-task protection level to maximize system-wide protection (§V-D).

Implementation and Evaluation: We have implemented a prototype of ParCSPI, including a partial context-sensitive data-flow analysis tool, a customized LLVM-based compiler that leverages efficient hardware features such as PA and Arm branch target identification (BTI). We validate the system using real-time CPS applications and benchmarks, and compare it to full context sensitivity on CPS: the WCET runtime overhead is reduced by 163% while still achieving 62% protection. Finally, we evaluate our MILP optimizer to determine the trade-offs between security and performance for real-time task sets and explore how solving time affects the determined security policy.

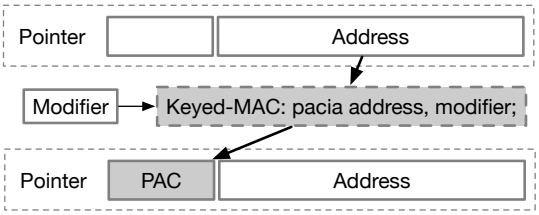

Fig. 1: Pointer authentication.

In summary, we have made the following contributions:

- We propose partial context-sensitive pointer integrity in §III, with the goal to maximize security protection for the cyber-physical system in the intended environment, by prioritizing coverage and context to reduce the equivalence class size for both data and code pointer integrity.
- We propose a real-time model for the enforcement of ParCSPI to balance the trade-off between real-time constraints and context granularity in §V. The real-time model accepts a user-defined security policy trade-off space as input and can automatically optimize the policy security while satisfying real-time constraints.
- We implement a prototype[1] of ParCSPI using a customized SVF-based tool and a LLVM-based compiler in §VI. We evaluated ParCSPI prototype on Armv8-A platforms on both real-time benchmarks and CPS to show feasibility in §VII-A and § VII-B.
- We develop in §V-D, and evaluate in §VII-C, an MILP response-time analysis that allows for system-wide ParCSPI optimization of the appropriate context(s) to maintain and check for each task while maximizing system-wide security.

II. Background

A. Arm Pointer Authentication

Pointer integrity is a security primitive aiming to ensure that pointer targets are not maliciously modified. Pointer Authentication (PA) is a security extension on Arm designed to enforce pointer integrity by signing the pointer at its creation and authenticating it whenever it is dereferenced. Figure 1 shows the workflow of the PA feature on AArch64 (the 64-bit architecture of the Arm architecture family). Specifically, PA works by adding instructions for creating and authenticating pointer authentication codes (PAC), where the PAC is a keyed message authentication code (MAC) calculated with a

[1]The source code is available on the website: https://parcspi.github.io/.

```
01: void func(int* ptr){
02:    int* p=ptr;
03:    ...//attacker reuses p and PAC generated...
04:       //under ctx2 with the one generated under ctx1
05:    cout<<*p<<endl;
06: }
07: func(&a);//ctx1
08: func(&b);//ctx2
```

Fig. 2: An example of pointer-reuse attack.

modifier, the pointer value, and a secret key stored in a special register accessible only by the kernel. During pointer creation, a PAC is generated and stored in (insecure) memory for pointer signing and validation. During the validation process, a new PAC is calculated, and the pointer is considered valid only if the two PACs match. As a result, since the PAC is a keyed MAC on the pointer value, which can only be calculated by the instrumented PA instructions, PA can prevent arbitrary modification of the pointer value.

B. Pointer-Reuse Attacks Against Pointer Integrity

Although PA can prevent arbitrary pointer corruption, an attacker can still reuse authenticated pointers (including the pointer values, modifiers, and PAC values). An example of a pointer-reuse attack [10] is shown in Fig. 2. The PAC for the pointer is calculated twice with different targets at line 2 under different contexts, i.e., function call sites `ctx1` and `ctx2`. The calculated PAC and the pointers are stored in insecure memory. At line 5, when using the pointers, both the pointer and the PAC are fetched from the insecure memory and go through pointer integrity checks. However, an attacker at line 3 can obtain the authenticated pointers (including the pointer values, modifiers, and PAC values) during the first invocation under context `ctx1`, and reuse them later during the second invocation under context `ctx2` to alter the pointer targets. This leads to alteration of the pointer targets without being detected by basic pointer integrity checks because the combination of the pointer value, modifier, and PAC is still valid.

C. Context Sensitivity

The reason behind the pointer-reuse attack is the over-approximation nature of static analysis, which makes the set of targets considered valid larger than the actual valid set of targets. This set of targets considered valid is referred to as the equivalence class (EC). A smaller EC contains fewer false negatives (i.e., targets considered valid but actually invalid), indicating better precision and thus stronger security protection. To increase protection precision, context sensitivity—such as callsite, path, and heap context sensitivity [12], [13], [14]—can be leveraged to narrow the EC. The key idea is to use runtime execution context information to further eliminate infeasible targets. As discussed in §III-C, to provide finer-grained protection against pointer-reuse attacks, context sensitivity is applied to pointer integrity to narrow the EC of pointer-integrity checks.

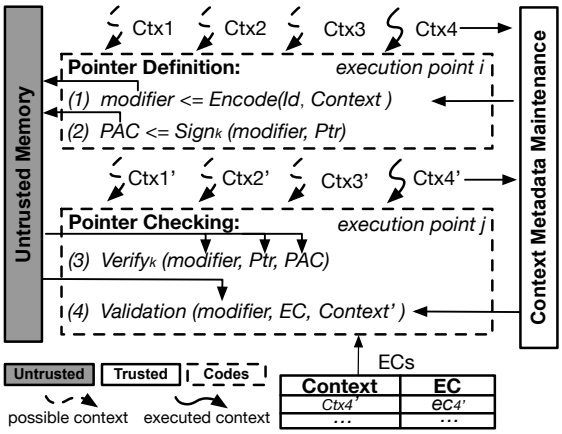

Fig. 3: Context-sensitive pointer integrity.

III. Motivation of Partial Context Sensitivity

In this section, we illustrate the workflow of context sensitivity in defending against pointer-reuse attacks, and the motivation for applying partial context sensitivity to enable a flexible trade-off between security and runtime performance.

A. Context-Sensitive Pointer Integrity

Pointer integrity is vulnerable to pointer-reuse attacks, where attackers can reuse authenticated pointers (including the pointer values, modifiers, and PAC values). To mitigate this attack, existing work has utilized runtime context to restrict the set of pointers considered valid [8], [9], [16]. The workflow of this process is shown in Fig. 3, including two key components: encoding context into the modifiers and using context to validate the pointer value. This process and figure are further explained below.

First, at the pointer-creation location, the runtime context and the compile-time-assigned location ID are encoded into the pointer modifiers (Step 1). The encoded runtime context can be used to capture the *temporal* information, representing when this pointer is created; the compile-time-assigned location ID can be viewed as *spatial* information, representing where this pointer is created. Then, in Step 2, such *temporal* and *spatial* information is bound to the pointer by signing the modifier with the pointer value to create a signature value, i.e., PAC. Afterwards, the PAC value, modifier, and pointer value are stored in insecure memory.

Upon usage of the pointer, the PAC value, modifier, and pointer value are retrieved from non-secure memory. Rather than using a static modifier, context-sensitive modifier for the specific dereference is looked up from a pre-computed table that maps context and dereference location into a unique modifier that represents all the possible valid defs within the given context (Step 3). As a result, the previous example of pointer-reuse attack, shown in Fig. 2, which can bypass basic pointer integrity checks, can be detected with the use of context. Specifically, execution context (e.g., calling context reflected as call site in this example) is leveraged to narrow

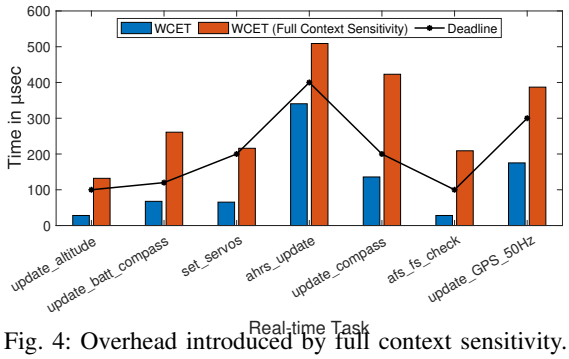

Fig. 4: Overhead introduced by full context sensitivity.

down the EC. The call site can be recorded as runtime context at line 7 and line 8.

B. High Overhead of Full Context Sensitivity.

However, using full context for runtime maintenance and pointer-integrity checks can significantly increase overhead. Figure 4 shows the WCET expansion when ArduPilot [15], an autopilot system, uses full context-sensitive pointer integrity to defend against pointer-reuse attacks. During runtime, the full calling context, recorded as function call site, is used at each pointer use point to check its validity. According to our preliminary results of utilizing full context, the new WCET can increase by 273% and exceed the deadline of its real-time tasks, which can be detrimental to real-time CPS (e.g., causing crashes). As a result, system designers have to face the trade-off between lightweight security primitives with limited protection or costly security primitives that can provide strong protection but will significantly impact real-time performance.

C. Partial Context-sensitive Pointer Integrity

Due to the high cost of full context sensitivity, existing work either uses lightweight context or protects only a portion of pointers for runtime efficiency [8], [10], [16]. Existing approaches lack support for enabling a flexible trade-off between security and real-time performance. We propose using partial context-sensitive pointer integrity (ParCSPI) for different computation and security demands, such that only a subset of program locations is associated with context and only a subset of pointer-use points is checked according to a precomputed security policy under partial context sensitivity.

IV. Threat Model and System Goals

We assume there are memory-corruption vulnerabilities that an attacker can leverage to perform arbitrary memory reads/writes. In line with the threat models from previous work [9], [10], [17], we do not assume there is dedicated secure memory storage in unprivileged mode. We assume code injection is prevented using existing memory-protection technology such as a Memory Management Unit (MMU), and all instrumented code, kernel stack and hardware stack are trustworthy. We assume that the attacker cannot infer the PA keys, as they are in registers not directly readable from user space. We assume the program to be protected does not contain PA-related instructions, and the generated PAC values cannot be brute-forced. We only consider attacks that exploit memory-corruption vulnerabilities to launch code-reuse attacks; other attack vectors such as hardware attacks [18] and side-channel attacks [19] are out of scope.

System goals: The system goals are described as follows:

Goal 1: Detecting data- and code-pointer-reuse attacks: Even when pointer integrity is deployed, attackers may alter pointer values by reusing previously authenticated pointers. ParCSPI aims to detect reuse attacks by enforcing statically computed context-sensitive def-use relationships on pointers using runtime execution context. Like previous work on pointer integrity [8], [9], [10], [20], [21], we exclude protection against pure non-pointer data-corruption attacks from the system goals. Since our protection is probabilistic and subject to real-time constraints, the prevention aims to protect those that are prioritized based on the intended operating environment.

Goal 2: Meeting the security goal under real-time constraints: Since program execution can generate rich context information, naïvely using execution context to detect reuse attacks can be expensive. Since high runtime overhead will delay tasks from their deadlines, causing serious consequences [22], [23], ParCSPI provides a policy-based framework to enable a flexible trade-off between context granularity and real-time performance.

V. ParCSPI Design

A. System Overview

The goal of ParCSPI is to enable a flexible trade-off between context/coverage of attack detection and performance under different real-time requirements. The overview of ParCSPI is shown in Figure 5, with three key steps. First, to detect reuse attacks while enabling a flexible security and performance trade-off, ParCSPI introduces partial context to pointer integrity. To reason about the provided security, ParCSPI develops a set of rules to derive the corresponding equivalence class under the partial context. To ensure coherency of the defense, it is crucial that the context selection at various pointer definition and usage points remains consistent. Inconsistent protection, particularly if initial pointer operations are safeguarded with a coarse-grained context, can render subsequent fine-grained protections ineffective. This inconsistency allows reuse attacks to exploit the initial coarse-grained safeguards, thereby bypassing later protections. Second, the problem of selecting pointers and contexts is formulated as a system-wide optimization problem, with the goal of maximizing security protection while meeting real-time constraints. Finally, ParCSPI instruments the program(s) according to the generated security policy. During runtime, execution context is recorded and maintained, and the pointer def and use are enforced to follow the statically computed context-sensitive def-use relations to detect reuse attacks.

B. Security Policy under Partial Context Sensitivity

As discussed in §III-C, the def-use relationships of pointers are validated during runtime according to a pre-defined se-


```
01: void func(int* p){ …//reuse attacks against p
02:    cout<<*p<<endl; }
03: void func_q1(int* q1){ …//reuse attacks against q1
04:   __CTX_RECORD__; func(q1);//p's context recorded }
05: void func_q2(int* q2){ …//reuse attacks against q2
06:   __CTX_RECORD__; func(q2);//p's context recorded }

07: func_q1(&a); func_q1(&b);//q1's context NOT recorded
08: func_q2(&c); func_q2(&d);//q2's context NOT recorded
```

Fig. 6: Example of security coherence.

To illustrate this security-coherence issue, consider the example shown in Fig. 6. In this example, the def-use enforcement of pointer `p` can distinguish its two definitions at lines 4 and 6 with its recorded context, such that a reuse attack at line 1, which substitutes between these two definitions, can be detected. However, since its source pointers, `q1` and `q2` in

this case, are not protected against reuse attacks, reuse attacks at lines 3 and 5 targeting p's source pointers q1 and q2 can indirectly alter the pointer p.

As a result, to ensure the consistency between the analyzed and achieved security guarantees, when protecting a pointer, ParCSPI automatically includes all its dependent pointers into partial context-sensitive pointer integrity checks at the same context level as the pointer's partial context-sensitive def-use equivalence class.

C. Per-Task Optimization

A security policy includes: 1) which pointers to check and 2) the context granularity for checks. Different security policies exhibit different security and real-time impacts. The goal of per-task optimization is to generate the optimal security policy to maximize security protection under a given WCET expansion constraint. The per-task WCET expansion constraint can be determined by system-wide optimization, as discussed in §V-D, to maximize system-wide security while ensuring schedulability.

Security quantification: We follow the existing work on context sensitivity [12] by using the largest EC size of a pointer, derived from Eq. 3, to quantitatively measure the protection granularity. Even though a common approach towards EC minimization considers all elements in the EC with equal security criticality, it also supports the incorporation of domain knowledge by tuning the weights. The security score of a single pointer l under a security policy π is denoted as $S_l(\pi)$ and is defined as the ratio of the full context sensitivity to partial context sensitivity:

$$S_l(\pi) = \begin{cases} \frac{\max\limits_{c \in C} |\langle c, l \rangle_t|}{\max\limits_{c^\pi \in C^\pi} |\langle c^\pi, l \rangle_t|}, \text{ if } l \text{ is selected by } \pi. \\ 0, \text{ otherwise.} \end{cases} \tag{4}$$

where $\max\limits_{c \in C} |\langle c, l \rangle_t|$ is a constant representing the EC size of full context, and $\max\limits_{c^\pi \in C^\pi} |\langle c^\pi, l \rangle_t|$ is the EC size under the partial context of the security policy π. Since the full context always has the smallest EC size (as it provides the strongest protection), S_l is therefore in the range $[0, 1]$. A higher score means a smaller EC size, thus finer protection granularity. Moreover, when a pointer is not selected for protection, the security score becomes 0 because the pointer can be arbitrarily modified, indicating an infinitely large EC.

With the security quantification of a single pointer, the security quantification for the entire task, denoted as $\mathcal{S}(\pi)$, is the (weighted) average security score of individual pointers:

$$\mathcal{S}(\pi) = \sum_l w_l * S_l(\pi) \tag{5}$$

where w_l is the user-specified security preference weight with $\sum_l w_l = 1$, to specify application semantics into the security quantification. By default, pointers have equal preference. When assessing mission-aware security optimization with known intended environments, w_l can also be determined by the probability of hitting the statement given an input distribution.

Optimization formulation. With the security quantification as described above, the optimization problem can be formulated as finding the policy π that maximizes the security score $\mathcal{S}(\pi)$ with the WCET expansion percentage ϵ as constraint:

$$\max_\pi \mathcal{S}(\pi) \text{ w.r.t } \Delta(\pi) \leq \epsilon \cdot WCET_{ori} \tag{6}$$

where $\epsilon \cdot WCET_{ori}$ is the maximum allowed WCET expansion and $\Delta(\pi)$ is the increased WCET, which can be derived using existing timing analysis tools. The insight behind this optimization is to select the context and conduct checks on pointers that can best reduce the EC size while achieving minimal impact on WCET expansion. Contexts that do not contribute to EC reduction are not chosen, and pointers and contexts that are not on the WCEP are more likely to be selected since they do not impose additional WCET expansion.

The optimization problem is solved in two steps. First, a one-time preprocessing step computes the security score of individual pointers under different partial contexts. Since the optimization requires measuring the WCET impact, which is often conducted by third-party timing analysis tools [27], obtaining a closed-form solution can be difficult. In the second step, we use a greedy algorithm to gradually select the contexts to maintain and the pointers to protect based on their impacts on security and WCET overhead. More details on the optimization can be found in §VI. The final output of the per-task optimization process is a table of WCET expansions and optimal security scores with corresponding security policies. This table is then used for system-wide optimization to determine the optimal per-task security policy that best contributes to system-wide security.

D. System-Wide Optimization

We exploit the elasticity of ParCSPI in order to balance security with runtime. We wish to choose an optimal policy under real-time constraints in order to maximize the total security score without violating schedulability. Thus, we design a Mixed Integer Linear Programming (MILP) optimization problem that allows us to maximize the security while guaranteeing that the task system remains schedulable.

Task Model: Assume that we have a set $\mathcal{T}$ of n tasks: $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$. Each $\tau_i \in \mathcal{T}$ has an original cost of C_i^0, a period of T_i, and a deadline of D_i. We assume that we are able to generate a trade-off table for each task τ_i. The trade-off table entries represent security score and WCET expansion pairs. Each row k in the table contains an ϵ_{ik} and S_{ik} pair for some policy π_{ik} for τ_i (i.e., derived from different values of ϵ in Eq. 6). Let the number of entries in this table be m_i. We will call the cost of task $\tau_i \in \mathcal{T}$ with the overhead of the selected security policy C_i.

MILP Formulation: In order to represent choosing a policy we define a boolean variable $\mathbf{X}_{ik}$ for each π_{ik}. We declare

that $\mathbf{X}_{ik} = 1$ implies that τ_i is using π_{ik} and so achieving a security score of S_{ik} and incurring an overhead of ϵ_{ik}.

Thus, the optimization criterion is as follows:

$$\text{maximize} \sum_{i=1}^{n} \sum_{k=1}^{m_i} \mathbf{X}_{ik} S_{ik} \tag{7}$$

However we must keep the task system schedulable.

A task can only use one policy at a time. We therefore introduce the constraint:

$$\forall \tau_i \in \mathcal{T} : \sum_{k=1}^{m_i} \mathbf{X}_{ik} \leq 1 \tag{8}$$

This enforces that a maximum of one $\mathbf{X}_i$ is set to 1.

In order to require that all tasks must remain schedulable we can represent schedulability as an Integer Linear Program as described by Baruah & Ekberg [28].

We assume deadline-monotonic (DM) scheduling on a single processor and that we therefore have sorted our task set by relative deadlines so that the priority of τ_i is greater than the priority of τ_{i+1}.

The ILP representation of response-time analysis in [28] proceeds as follows:

For each $\tau_i \in \mathcal{T}$ we define a new variable $\mathbf{R}_i$ that represents the response time of τ_i. In order for the task system to remain schedulable, each task must meet its deadline D_i, so we introduce a new constraint:

$$\forall \tau_i \in \mathcal{T} : \mathbf{R}_i \leq D_i \tag{9}$$

We introduce a new variable to represent the $\lceil \mathbf{R}_i / T_j \rceil$ term from standard response time analysis. For each task τ_i and each higher-priority task τ_j, we create a new non-negative integer variable $\mathbf{Z}_{ij}$ with the constraint:

$$\mathbf{Z}_{ij} \geq \left(\frac{\mathbf{R}_i}{T_j} \right) \tag{10}$$

Finally, in order to represent the mathematical model of schedulability Baruah & Ekberg use the following constraint:

$$\forall \tau_i \in \mathcal{T} : C_i + \sum_{j=1}^{i-1} \mathbf{Z}_{ij} C_j \leq \mathbf{R}_i \tag{11}$$

However, this assumes that the cost for each task is a constant. We violate that assumption since the overhead of each possible policy is different. Thus we differentiate between the original cost of τ_i which we notate as C_i^0 and the actual cost of τ_i which is C_i. We define C_i in terms of C_i^0 as:

$$C_i = C_i^0 \left(1 + \sum_{k=1}^{m_i} (\mathbf{X}_{ik} \epsilon_{ik}) \right) \tag{12}$$

We note that since only one X_i can be true at any time, and each X_i is a binary variable, the original cost will only be altered by the chosen ϵ_i.

For ease of presentation, we will define an intermediate variable H_i that represents the effect of higher priority tasks on the response time of τ_i. In Constraint (11), this is equivalent to $\sum_{j=1}^{i-1} \mathbf{Z}_{ij} \times C_j$, however given the definition of C in terms of C^0, we can express this as:

$$H_i = \sum_{j=1}^{i-1} [(\mathbf{Z}_{ij} C_j^0) + C_j^0 \sum_{k=1}^{m_j} (\mathbf{Z}_{ij} \mathbf{X}_{jk} \epsilon_{jk})] \tag{13}$$

Note that we have distributed $\mathbf{Z}_{ij}$ into the summation. Importantly, we can see that there is the multiplication of two variables $\mathbf{Z}_{ij}\mathbf{X}_{jk}$ making the constraint seemingly non-linear.

We get around this problem by leveraging the observation that the result of the multiplication is either $\mathbf{Z}_{ij}$ or 0 since X_{jk} is a boolean. Thus, we can use well-known integer-programming techniques for linearizing products. We define new variables $\mathbf{Y}_{ijk}$ to represent the product $\mathbf{Z}_{ij}\mathbf{X}_{jk}$. Additionally, we define a constant $Z_{ij}^U = \lceil T_i / C_j^0 \rceil$ as an upper bound for $\mathbf{Z}_{ij}$. Note $\mathbf{Z}_{ij}$ represents the number of times that τ_j can run during the response time of τ_i. Since we require that everything remain schedulable, and τ_j will not run more often than its period, we can upper bound $\mathbf{Z}_{ij}$ with the number of times that C_j^0 could fit into T_i. We define the following constraints for each $\mathbf{Y}_{ijk}$:

$$\begin{aligned} 0 &\leq \mathbf{Y}_{ijk} \\ \mathbf{Y}_{ijk} &\leq \mathbf{X}_{jk} Z_{ij}^U \\ 0 &\leq \mathbf{Z}_{ij} - \mathbf{Y}_{ijk} \\ \mathbf{Z}_{ij} - \mathbf{Y}_{ijk} &\leq (1 - \mathbf{X}_{jk}) Z_{ij}^U \end{aligned} \tag{14}$$

The first two inequalities ensure that $\mathbf{Y}_{ijk} = 0$ when $\mathbf{X}_{jk} = 0$. The second two inequalities ensure that $\mathbf{Y}_{ijk} = \mathbf{Z}_{ij}$ when $\mathbf{X}_{jk} = 1$. We can now use $\mathbf{Y}_{ijk}$ to update our definition of H_i so that it is linear:

$$H_i = \sum_{j=1}^{i-1} \left((\mathbf{Z}_{ij} C_j^0) + C_j^0 \sum_{k=1}^{m_j} (\mathbf{Y}_{ijk} \epsilon_{jk}) \right) \tag{15}$$

Finally, we define our schedulability constraint as:

$$\forall \tau_i \in \mathcal{T} : C_i + H_i \leq R_i \tag{16}$$

Subject to the defined constraints we have defined a MILP optimization problem where we choose policies that maximize the sum of the security scores for each task while guaranteeing that the task set remains schedulable.

VI. Implementation

ParCSPI is built on Armv8-A platform on Raspberry Pi 3 Model B with callsite as context, and the implementation consists of three parts: security policy generation, context metadata protection, and program instrumentation.

Security-policy generation. To choose the security policy for a task, both the security and runtime impact of different security instrumentations, such as different levels of partial context-sensitive checks, are calculated. To calculate the security score for each partial context-sensitive pointer integrity check, ParCSPI builds a customized static program analyzer

Algorithm 1: Per-task optimization

```
   Input: I, S, ϵ, P
   Output: I_s                          // selected instrumentation
1  I_s ← {}
2  while I not empty do
3      i_s ← max_{i∈I}{ratio(i, I_s)}
4      if K({i_s} ∪ I_s ∪ P)) ≤ (1 + ϵ) * WCET_ori) then
5          I_s ← {i_s} ∪ I_s                          // select i_s
6          I.pop(i_s)
7      else
8          break
9  return I_s
10 Function ratio(i, I_s):
       /* calculate the ratio of (↑ Sec)/(↑ Cost)     */
11     r ← [S(I_s ∪ {i}) − S(I_s)]/[K(I_s ∪ P ∪ {i}) − K(I_s ∪ P)]
12     return r
```

Note: I: candidate security-instrumentation points, S: security score, ϵ: WCET overhead threshold, P: original program, $\mathcal{K}$: WCET measurement, I_s: selected instrumentation points.

based on SVF [24], [26], a state-of-the-art static analysis tool, to conduct partial context-sensitive analysis. The static program analyzer first performs full context-sensitive data-flow analysis and then derives the def-use equivalence class under partial context from the analysis results of full context sensitivity, as described in §V-B. From this, the security score, S, for each pointer under partial context is calculated.

With the calculated security scores of individual pointers and partial contexts, ParCSPI uses a greedy algorithm to find an approximation of the optimal security policy, I_s, from all candidate security-instrumentation points (i.e., pointer checks and context maintenance operations), I. As shown in Alg.1, the algorithm iteratively computes the ratio between the increased security score and the increased WCET overhead for each security-instrumentation candidate. Since a larger ratio value indicates increased security with minimal WCET impact, ParCSPI gradually selects the security instrumentation with the maximal ratio (line 3) until the WCET threshold ϵ is exceeded. The final computed security policy and security score under different WCET thresholds are then used for system-wide optimization to determine time allocation for individual tasks.

Context Metadata Protection Given that the recorded context is stored in untrusted memory, to prevent attackers from corrupting the context metadata, we use PA to sign the recorded context into a hash chain and store the final hash value in a reserved register, similar to the existing work on protecting the return address using a hash chain [9].

$$\begin{aligned} auth_{i-1} &\leftarrow R_v \\ auth_i &\leftarrow \mathcal{P}(auth_{i-1}, c) \\ R_v &\leftarrow auth_i \end{aligned} \tag{17}$$

where $\mathcal{P}$ is the PA signing process, c is the recorded context, and $auth_{i-1}$ and $auth_i$ are the previous and updated final hash chain values that are stored in the reserved register R_v. To verify the calling context metadata, ParCSPI first loads the value from R_v, along with $auth_{i-1}$ and the context metadata c from the untrusted memory. Then, ParCSPI performs PA verification $\mathcal{A}(R_v, auth_{i-1}, c)$, where the verification $\mathcal{A}$ triggers an exception if $R_v \neq \mathcal{P}(auth_{i-1}, c)$. Thus, this process can verify the integrity of the context metadata because any malicious modification of c or $auth_{i-1}$ will be detected by the last verification. Since an attacker cannot manipulate the value in R_v, any metadata modification will generate a mismatch that is detected.

Program Instrumentation. To conduct pointer integrity checks against runtime context, the equivalence classes are encoded as a lookup table, and the table is protected from attacker modification by configuring the memory region as read-only using MMU hardware, and security instrumentation is inserted into the protected code by customized LLVM passes. However, an attacker can trigger malicious control flow to jump directly into the middle of the instrumented code, thereby bypassing security enforcement. To prevent such attacks on instrumented security primitives, ParCSPI utilizes the Branch Target Identification (BTI) security hardware feature on Arm. BTI ensures that all indirect jumps can only target BTI-specific instructions that are inserted into the program code. Therefore, BTI serves as an efficient but coarse-grained control-flow integrity (CFI) enforcement. To this end, ParCSPI inserts BTI instructions at function entries and after call sites to prevent attackers from executing injected code in the middle.

VII. Evaluation

The evaluation focuses on measuring the ability of ParCSPI to balance security and performance in real-time systems. We evaluate ParCSPI using both real-time benchmarks and cyber-physical applications on the Arm Cortex-A platform. ParCSPI is implemented using the hardware features PA and BTI; however, to our knowledge, there is no publicly available embedded platform that supports both features. Therefore, instruction analogs are used to assess the performance overhead. Instruction analogs are a series of instructions that consume the same CPU cycles and the same memory footprint as the PA/BTI instructions but do not perform an actual check [11], [16], [29]. For PA, we adopt the same PA analogs as PAL [16] such that one PA operation equals to seven exclusive-or (*EOR*) instructions. As for BTI analogs, we analog a BTI instruction as one *NOP* instruction. We analog BTI in this way because it checks a register bit field *PSTATE.BTYPE* [30], [31], which we assume is efficient for hardware-implemented BTI. The expanded WCET is then measured with the replaced instructions.

A. Security and Performance Trade-offs

To demonstrate the trade-offs between the provided security and the corresponding WCET impacts, we conducted measurements on the BEEBS embedded benchmark [32] treating all pointers with equal preference. Table I shows the basic pointer integrity WCET impact as well as ParCSPI's security protection (quantified as security score in Eq. 5) under different WCET expansions, while Fig. 7 illustrates the detailed relationship between the security protection and WCET expansion trade-off.

TABLE I: Security score under different WCET expansions.

Program	sg_dllist	mergesort	huffbench	ndes	picojpeg	sg_queue	qrduino	levenshtein	cubic	st	sg_hashtable	edn	wikisort	stb_perlin	fir
$\mathcal{S}_\pi/\epsilon_\pi$	0.02/138%	0.33/46%	0.16/10%	0.15/23%	0.18/39%	0.05/1%	0.15/11%	0.4/17%	0.21/1%	0.3/18%	0.05/115%	0.11/26%	0.09/58%	0.5/11%	0.07/48%
$\epsilon = 10\%$	0.81	0.64	0.76	0.52	0.70	1.00	0.95	0.70	1.00	0.58	0.75	0.58	0.88	0.25	0.79
$\epsilon = 50\%$	0.85	0.84	1.00	0.74	0.81	1.00	1.00	0.90	1.00	0.79	0.81	0.92	0.92	0.75	0.86
$\epsilon = 100\%$	0.91	0.90	1.00	0.93	0.89	1.00	1.00	0.97	1.00	0.84	0.88	0.97	0.95	1.00	1.00
Program	**slre**	**dijkstra**	**insertsort**	**bubblesort**	**fasta**	**frac**	**whetstone**	**sg_rbtree**	**arcfour**	**listsort**	**rijndael**	**nbody**	**matmult_int**	**aes**	
$\mathcal{S}_\pi/\epsilon_\pi$	0.07/56%	0.18/26%	0.04/121%	0.29/44%	0.34/20%	0.06/19%	0.16/6%	0.02/93%	0.1/45%	0.05/129%	0.03/10%	0.13/20%	0.3/26%	0.04/40%	
$\epsilon = 10\%$	0.69	0.31	0.85	0.14	0.86	0.18	0.48	0.83	0.52	0.22	0.74	0.33	0.70	0.39	
$\epsilon = 50\%$	0.84	0.91	0.94	0.29	0.90	0.59	1.00	0.91	0.71	0.46	0.99	0.80	0.70	0.81	
$\epsilon = 100\%$	0.93	1.00	0.96	0.43	0.93	0.88	1.00	0.95	0.81	0.73	1.00	0.93	0.80	0.96	

$\mathcal{S}_\pi$: security score of basic pointer integrity. ϵ_π: WCET expansion of basic pointer integrity. ϵ: WCET expansion of ParCSPI.

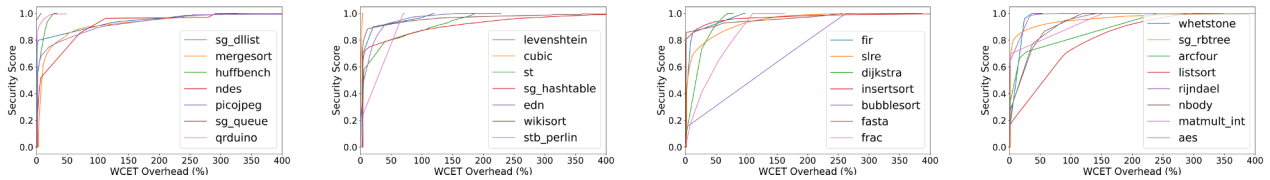

Fig. 7: Security and performance tradeoff.

With only 10% WCET expansion, ParCSPI can achieve an average security protection of 0.62, while basic pointer integrity incurs 41% WCET overhead with only 0.16 security protection. This is due to the per-task optimization engine that selects the maintained context and protected pointers, which can achieve the largest security protection with minimal WCET overhead. On the contrary, due to the insufficient context information of basic pointer integrity, pointer targets are considered valid if the PAC generation is valid. Attackers can reuse any authenticated pointers to alter the pointer targets, incurring a large EC size compared to full context sensitivity, thus achieving a low security score.

There is a small, fixed WCET impact from BTI (1.05% on average) to provide basic CFI protection. One observation is that ParCSPI can still provide protection with this negligible fixed WCET impact because the instrumented security code that is not on the WCEP can provide protection without impacting WCET. The security score increases rapidly when ϵ starts to grow. This is because most pointers start to be protected by pointer integrity with lightweight context, which is cost-efficient and contributes the most to the security quantification. Afterward, ParCSPI reaches full context-sensitive protection (i.e., $\mathcal{S}_\pi = 1$) with an average WCET overhead of 173%.

Moreover, ParCSPI provides different levels of protection for different programs under the same WCET expansions due to several reasons. First, the distribution of pointers differs among programs. Examples include benchmark programs *sg_dllst* and *cubic*, where the pointers to be protected in the former concentrate more on the WCEP, requiring larger computational resources to achieve the same level of security protection. The second reason is the sensitivity of programs to context. Since the security score measures the distance between full context and partial context, programs that rely more on context to narrow the EC and those with context concentrating more on the WCEP exhibit a larger WCET expansion to achieve a similar level as full context.

TABLE II: Applying ParCSPI to real-time CPSs.

Application	WCET (μs)	Deadline (μs)	Policy	Security Score
ArduCopter	67	120	π_0	0.64
			π_1	0.51
ArduRover	340	400	π_0	0.39
			π_1	0.34
ArduPlane	175	300	π_0	0.65
			π_1	0.54

Note: π_0: default policy where each pointer has equal preference. π_1: code pointers have 2 times the preference weight of data pointers.

B. Applying ParCSPI to Real-Time CPSs

To demonstrate the effectiveness of applying ParCSPI to real-world applications, we conducted measurements on three real-time CPSs from Ardupilot, an open-source self-driving project [15]: ArduCopter, ArduRover, and ArduPlane. The tasks selected are *update_batt_compass*, *ahrs_update*, and *update_GPS_50Hz* correspondingly. The task deadlines, obtained from the source code, are regarded as the WCET expansion constraints. The evaluation was conducted under two user-specified policies with different preferences on the pointers to be protected, where π_0 is the default policy treating all pointers with equal preference, and π_1 assigns the code pointers with higher preference than data pointers.

Quantitative analysis. Table II shows the provided security score under the real-time constraints. The security score is calculated by treating all pointers with equal preference weights. The results show that ParCSPI can still provide 51% security protection coverage within the allowable WCET expansion. In general, the default policy achieves a higher security score because optimization under policy definitions with user-specific preference has an additional constraint than full automatic approach: the optimal policy tends to prioritize the code pointers even the gained security (EC reduction) is less than the data pointers for the same time budget.

Case study. To assess the security gain of ParCSPI, a qualitative case study was conducted. In autonomous vehicles, a

```
01: void Rover::radio_failsafe_check(uint16_t pwm){
02:     if (AP_HAL::millis() - failsafe.last_valid_rc_ms > 500) {
03:         failed = true;} ...
04:     failsafe_trigger(FAILSAFE_EVENT_THROTTLE, "Radio", failed);} //callsite 1
05: void Rover::gcs_failsafe_check(void){...
06:     failsafe_trigger(FAILSAFE_EVENT_GCS, "GCS", do_failsafe);} //callsite 0
07: void Rover::failsafe_trigger(uint8_t failsafe_type, const char* type_str, bool on){...
08:     float ret_waypoint = [10];
09:     read_block(ret_waypoint, storage_path); //the attacker substitutes pointer from
'type_str="GCS"' to 'type_str="Radio"'
10:     return_home(ret_waypoint); //guide vehical to return home
11:     if ((failsafe.triggered == 0) && (failsafe.bits != 0)) {...
12:         gcs().send_text(MAV_SEVERITY_WARNING, "%s Failsafe", type_str);...}...}
13: void GCS::send_text(MAV_SEVERITY severity, const char *fmt, char* type_str){...
14:     pointer_dereference_check(type_str);...}
```

Fig. 8: Code of the case study.

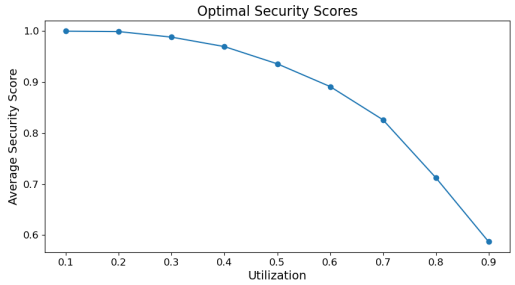

Fig. 9: Optimal average security scores per task vs total utilization of synthetic task sets.

fail-safe controller is often customized to keep autonomous vehicles safe under emergency conditions. In this case study, the fail-safe controller reads pre-stored home way points in storage and guides vehicle to return. We inserted a buffer-overflow vulnerability in the fail-safe controller within the code base of ArduRover. Specifically, reading home way points can overwrite the pointer *type_str* from "GCS" (ground control station) to "Radio" by a pointer-reuse attack as shown in Fig. 8. Then, function *send_text* inside function *failsafe_trigger* may send out the wrong message ("Radio Failsafe" instead of "GCS Failsafe"). The vehicle operator will not make emergency responses in time if the wrong message is received.

ParCSPI can detect this attack with partial context generation. Assuming that context is only partially generated in the protected program, as shown in the example. During offline context-sensitive analysis, the EC under the context callsite 0 for pointer dereferencing `type_str` is the set that only contains the pointer definition '`type_str`="GCS" '. Therefore, the above attack, which alters the pointer `type_str` from "GCS" to "Radio", can be detected during the pointer dereferencing checking inside function `send_text`. ParCSPI detects the above attack within the task's deadline constraint. However, under full context sensitivity, the task's WCET expansion is 143%, exceeding the deadline.

C. System-Wide Optimization Evaluation

In order to evaluate our system-wide optimization MILP implementation we present two experiments. The first experiment aims to determine the optimal security level of synthetic task sets given different original system-utilization levels. The second experiment evaluates the trade off between solver

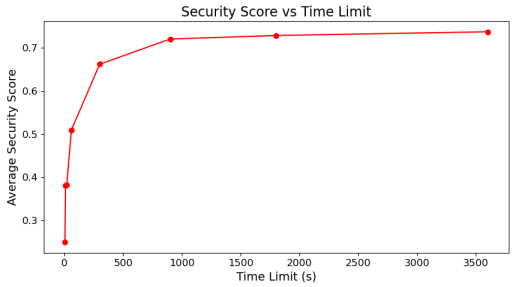

Fig. 10: Average security score found over time.

performance vs. execution time. We ran these experiments over a randomly generated synthetic task set assigning each task real-world security vs. overhead trade-offs determined from the BEEBS dataset [32] as shown in Figure 7.

We represented these trade-offs in a "trade off table" for each task. For our synthetic task-set generation, we utilized the Schedcat [33] TaskGenerator. We utilized the GNU Linear Programming Kit (GLPK) solver using the PuLP API on an 8 Core, 3.0GHz CPU with 16 GiB Memory.

For the optimal security level experiment, we ran the solver for each system utilization level under test. We considered total utilizations in $\{0.1, 0.2, \ldots, 0.9\}$. Task periods were generated by sampling uniformly among $[10, 100]$ms. This is equivalent to the built-in 'uni-moderate' distribution as used in [33], [34]. For the task utilizations, we choose uniformly random from $[0.1, 0.4]$. This is the built-in 'uni-medium' distribution.

We generated over 3000 task sets per utilization level. We discarded any task sets that were not schedulable without instrumentation. We randomly assigned a trade-off table to each task in the task set from the tables measured from the BEEBS benchmark. Finally we performed the optimization on each task set using a time limit of 1 hour per optimization. We averaged the security score per task for each task set and then averaged this per generated system utilization value. We graph the results of this in Figure 9.

We observe the average per-task security score decreases as the original system utilization increases. This is because in order to achieve higher security there is more overhead. Thus, when the original utilization is higher and there is less available time, we cannot achieve as high a security score.

Scalability. MILP is a known NP-complete problem, so we evaluated the scalability of our optimization. We found that for task systems with more than approximately 10 tasks, the execution time was often over one hour. However, MILP solvers can often quickly find *feasible* solutions, and much of the execution time can often be spent finding the optimal value. Therefore, we evaluated the solution quality over time that the solver is able to generate. We compared the solutions given by the solver after 5 seconds, 10 seconds, 20 seconds, 1 minute, 5 minutes, 15 minutes, 30 minutes, and 1 hour.

For this experiment we generated five unique task sets. We defined the maximum utilization to be 0.6, and used the same 'uni-moderate' period distribution. In this experiment we

TABLE III: Pointer integrity protection.

System	Target Attack						Not Req. Cust. HW	Not Req. Sec. Str.	RT Ada.
	Code Pointer				Data Pointer				
	Ret. Addr		Ind. Call						
	Arb. Mod	Reuse	Arb. Mod	Reuse	Arb. Mod	Reuse			
PACStack [9]	✓	✓					✓	✓	
CPI [21]	✓	✓	✓	✓			✓		
PARTS [10]	✓		✓		✓		✓	✓	
PACTight [8]	✓	✓	✓	✓			✓	✓	
PAC-PL [35]	✓		✓					✓	
PAL [16]	✓	✓	✓	✓			✓	✓	
ParCSPI	✓	✓	✓	✓	✓	✓	✓	✓	✓

Arb. Mod: Arbitrary Modification, Req. Cust. HW: Require Customized Hardware, Req. Sec. Str.: Require Dedicated Secure Storage, RT Ada.: Real-Time Adaptation

sampled the 'uni-light' distribution, which is drawn uniformly from $[0.001, 0.1]$ in order to increase the number of generated tasks. These settings were chosen to make the optimization more difficult is to force the number of tasks in the task set to be higher and thus to make the optimization more difficult. Additionally we limited the tasks to using a singe trade-off table for consistency. Specifically we used the table for the BEEBS `st` program.We ran the solver for each task set and time limit and plot the per-task average security score. The results of this experiment can be found in Figure 10.

We observe that while 1 hour may not be long enough to find the optimal solution, the average security score converges quickly, demonstrating that while the optimal solution may not be easily found, this formulation can be practically applied to improve the security posture of the system, even for larger task systems.

VIII. Related Work

Pointer integrity. As shown in Table III, various systems have been proposed to protect pointer integrity, and among them, only PAC-PL [35] requires customized hardware. To protect pointers against reuse attacks, existing work [21] assumes the existence of dedicated secure storage to protect pointers that are only accessible by trusted instrumentation code. In contrast, ParCSPI follows [9], [10], [16], [35] which do not impose such an assumption but rather assumes attackers can manipulate any memory values. Moreover, none of them can detect reuse attacks for both code and data pointers due to the high runtime overhead. This motivates the design of ParCSPI to be the first to protect both code and data pointers with real-time adaptation using partial context to achieve a flexible trade-off between security protection and real-time impact.

Pointer authentication-assisted security mechanisms. The PAC is considered as a MAC and is used to authenticate pointer or data values. Besides pointer integrity protection, this property has been leveraged in other security mechanisms such as memory safety and compartmentalization [11], [29], [36]. For example, PTAuth [36] leverages PA to protect temporal memory safety by checking temporal information of each object. PACMem [29] is a PA-based memory sanitizer that checks both temporal and spatial information of objects. HAKC [11] uses PA and Memory Tagging Extension (MTE) to do compartmentalization.

Control and data flow integrity: Control-flow integrity (CFI) aims at preventing attacks from subverting the control flow of the program by forcing the program execution to follow a predefined control-flow graph, which contains the intended program behavior information [12], [37], [38]. Context-sensitive CFI enhances this protection by leveraging context information recorded during runtime to provide more fine-grained control flow protection [12], [13], [14], [37], [39], [40]. While CFI hardens control flow, data flow can be protected with Data-flow Integrity (DFI), which ensures the runtime data flow follows the data-flow graph computed during static analysis [41], [17].

IX. Discussion and Limitations

Quantification of security: ParCSPI quantifies the security based on the EC size, a commonly used security metric in existing works [12], [42], [43]. This metric measures the gap between full and partial context-sensitive pointer integrity. While this quantification approach is fully automatic, it lacks application-specific semantic information, given that not all pointers are equally security critical in the application context. ParCSPI also allows users with domain knowledge to specify the criticality of each pointer to protect, and performs optimization accordingly.

Incorporating more context types. The consideration of execution context enhances the granularity of existing pointer integrity protection mechanisms by reducing the over-approximation in the security policy generated by static analysis of the program. However, the effectiveness of this reduction can vary depending on the type of context used. Consequently, even with the inclusion of context, equivalence classes may still contain multiple pointer targets, leaving some room for potential adversarial manipulation, even though it might be very difficult to meet the condition for exploitation. Besides call site context, context can take various forms, including stronger forms such as heap and path context sensitivity [13], which can be more effective in narrowing the EC and providing stronger security. To make other context types compatible with ParCSPI, it is necessary to design a new context-metadata maintenance scheme to protect the metadata, as well as a partial context-sensitive analysis algorithm for unsupported context models to reason about the provided security.

X. Conclusion

We have proposed a security primitive, partial context-sensitive pointer integrity, to defend against pointer-reuse attacks in real-time CPS. To enforce this security primitive, a system, ParCSPI, is proposed to provide a flexible trade-off between real-time performance and security protection. A prototype is developed on Armv8-A and evaluated on both real-time CPSs and benchmarks, demonstrating its effectiveness.

Acknowledgment

We thank the reviewers for their valuable feedback. We also want to express our gratitude to Yuhao Wu, Jack Zhai, Jinwen Wang, Ao Li, Dr. Ryan Gerdes, and Dr. Wenjing Lou for insightful discussions on earlier drafts of this paper. This work

was partially supported by the NSF (CNS-2038995, CNS-2038726, CNS-2141256, CNS-2238635, CPS-2229290, CNS-2403758), and ARO (W911NF-24-1-0155).

REFERENCES

[1] W. He, V. Zhao, O. Morkved, S. Siddiqui, E. Fernandes, J. Hester, and B. Ur, "Sok: Context sensing for access control in the adversarial home iot," in *IEEE European Symposium on Security and Privacy*, 2021.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[3] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium*, 2014.

[4] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow bending: On the effectiveness of Control-Flow integrity," in *24th USENIX Security Symposium*, 2015.

[5] A. Li and N. Zhang, "Data-flow availability: Achieving timing assurance in autonomous systems," in *18th USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[6] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *IEEE Symposium on Security and Privacy*, 2014.

[7] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy*, 2013.

[8] M. Ismail, A. Quach, C. Jelesnianski, Y. Jang, and C. Min, "Tightly seal your sensitive pointers with PACTight," *USENIX Security Symposium*, 2022.

[9] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: an authenticated call stack," in *30th USENIX Security Symposium*, 2021.

[10] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *28th USENIX Security Symposium*, 2019.

[11] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with HAKC," in *Network and Distributed System Security Symposium*, 2022.

[12] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium*, 2019.

[13] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of Path-Sensitive control security," in *26th USENIX Security Symposium*, 2017.

[14] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[15] "Ardupilot - a trusted, versatile, and open source autopilot system." https://ardupilot.org/, 2022.

[16] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication," in *31st USENIX Security Symposium*, 2022.

[17] Y. Wang, A. Li, J. Wang, S. Baruah, and N. Zhang, "Opportunistic data flow integrity for real-time cyber-physical systems using worst case execution time reservation," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, USENIX Association, 2024.

[18] M. A. Elmohr, H. Liao, and C. H. Gebotys, "EM fault injection on ARM and RISC-V," in *21st IEEE International Symposium on Quality Electronic Design*, 2020.

[19] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Trusense: Information leakage from trustzone," in *IEEE INFOCOM 2018-IEEE conference on computer communications*, 2018.

[20] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "Zerø: Zero-overhead resilient operation under pointer integrity attacks," in *ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2021.

[21] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018.

[22] J. Wang, Y. Wang, A. Li, Y. Xiao, R. Zhang, W. Lou, Y. T. Hou, and N. Zhang, "ARI: Attestation of real-time mission execution integrity," in *USENIX Security Symposium*, 2023.

[23] J. Wang, Y. Wang, and N. Zhang, "Secure and timely GPU execution in cyber-physical systems," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[24] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016.

[25] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016.

[26] Y. Sui and J. Xue, "Value-flow-based demand-driven pointer analysis for C and C++," *IEEE Transactions on Software Engineering*, 2018.

[27] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, "TimeWeaver: A tool for hybrid worst-case execution time analysis," in *19th International Workshop on Worst-Case Execution Time Analysis*, 2019.

[28] S. Baruah and P. Ekberg, "An ILP representation of Response Time Analysis." Available at https://research.engineering.wustl.edu/~baruah/Pubs.shtml, 2021.

[29] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "PACMem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[30] A. Holdings, "Arm architecture reference manual, ARMv8, for ARMv8-A architecture profile," *ARM, Cambridge, UK, White Paper*, 2019.

[31] "Armv8-m architecture reference manual," 2022.

[32] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[33] B. Brandenburg, "SchedCAT: The real-time scheduling toolkit." https://github.com/brandenburg/schedcat, 2024. Accessed: 2024-05-23.

[34] B. Brandenburg, *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, Chapel Hill, NC, August 2011.

[35] G. Serra, P. Fara, G. Cicero, F. Restuccia, and A. Biondi, "PAC-PL: Enabling control-flow integrity with pointer authentication in fpga soc platforms," in *IEEE 28th Real-Time and Embedded Technology and Applications Symposium*, 2022.

[36] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal memory safety via robust points-to authentication," in *30th USENIX Security Symposium*, 2021.

[37] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[38] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, 2009.

[39] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[40] Y. Wang, C. L. Mack, X. Tan, N. Zhang, Z. Zhao, S. Baruah, and B. C. Ward, "InsectACIDE: Debugger-based holistic asynchronous CFI for embedded system," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2024.

[41] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[42] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011.

[43] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," *ACM SIGPLAN Notices*, 2013.