ARI: Attestation of Real-time Mission Execution Integrity

Jinwen Wang*, Yujie Wang*, Ao Li*, Yang Xiao†,
Ruide Zhang‡, Wenjing Lou‡, Y. Thomas Hou ‡, Ning Zhang*

* *Washington University in St. Louis*
† *University of Kentucky*
‡ *Virginia Polytechnic Institute and State University*

Abstract

With the proliferation of autonomous safety-critical cyber-physical systems (CPS) in our daily life, their security is becoming ever more important. Remote attestation is a powerful mechanism to enable remote verification of system integrity. While recent developments have made it possible to efficiently attest IoT operations, autonomous systems that are built on top of real-time cyber-physical control loops and execute missions independently present new unique challenges.

In this paper, we formulate a new security property, Real-time Mission Execution Integrity (RMEI) to provide proof of correct and timely execution of the missions. While it is an attractive property, measuring it can incur prohibitive overhead for the real-time autonomous system. To tackle this challenge, we propose policy-based attestation of compartments to enable a trade-off between the level of details in measurement and runtime overhead. To further minimize the impact on real-time responsiveness, multiple techniques were developed to improve the performance, including customized software instrumentation and timing recovery through re-execution. We implemented a prototype of ARI and evaluated its performance on five CPS platforms. A user study involving 21 developers with different skill sets was conducted to understand the usability of our solution.

1 Introduction

With recent advances in computing, embedded systems are playing an increasingly important role in society from smart home appliances to automobiles. The commercialization of these autonomous systems is no longer a science fiction story, with the deployment of self-driving cars and drone delivery services [1]. Modern autonomous systems often need to support complex mission control functionalities using interconnected sensors, actuators, and processors [2]. Upon receiving a high-level command, the autonomous system has to operate through various uncertainties in the physical environment to accomplish the mission independently. Even though a cyber attack on these safety-critical autonomous systems can lead to catastrophic physical world consequences, existing back-end controllers often gain little visibility into the integrity of these remote autonomous systems.

Remote Attestation: Remote attestation, which allows a device to prove its integrity to a remote verifier, is a powerful security mechanism to ensure the correctness of a remote system [3–13]. In a remote attestation protocol, the prover needs to send his/her system measurements to the verifier, who can then confirm the integrity of the prover. While earlier remote attestation systems often focus on verification of static memory content [14–16], recent advances aim to also verify the runtime properties, such as control flow (CF) and data flow (DF) [3–5, 9–13]. A driving motivation behind runtime property verification is the capability to prove faithful execution of individual external commands [6–8].

Limitations for Real-time Autonomous CPS: Existing operation-based runtime property attestation systems [3,5] are often designed for Cloud-based IoT environments, where the Cloud-based controller is part of the control loop, remotely operating each sensing and actuation operation. However, attestation on autonomous systems faces two new challenges. First, system timing is important for real-time CPSs, as recently demonstrated by the newly emerged timing attacks [17–19], but none of the existing approaches considers the temporal property of the execution. Second, autonomous systems are often expected to operate independently to accomplish the mission, such as flying to a household location to drop off the package. Measurements have to be collected over the entire mission, even when there are no explicit commands from the remote controller.

Our Solution: To bridge this gap, we propose a new security property, Real-time Mission Execution Integrity (RMEI), and its attestation system, ARI (Attestation of Real-time mission execution Integrity) in this paper. Inspired by the idea of a flight recorder (black box) on an aircraft, instead of attesting the processing of a single command, ARI aims to provide an attestation of system integrity as well as its real-time property

throughout the entire mission. While the vision of mission attestation is attractive, it also brings new unique challenges.

First, there are more properties to measure for mission integrity. RMEI is principally derived from program behavior integrity via measurement of control flow and/or data flow events, as well as temporal characteristics via accurate time measurements for different stages of execution. The overhead can increase significantly with the additional instructions for fine-grained temporal property measurement, such as recording the timestamp of each control flow event. Second, there are substantially more events due to high-frequency control loops. For example, there are more than 3.1 million control flow events that need to be recorded per second in ArduCopter [20] controllers during a test flight mission. Furthermore, all these events also have to be passed to the secure domain for storage and signature, increasing runtime overhead significantly. However, the majority of autonomous systems are real-time systems in which missing a deadline for safety-critical tasks can be fatal. Thus, it is important that the attestation system does not significantly change the temporal profile of the system. Nonetheless, as detailed in Section 2, adapting existing attestation approaches naively, even without the addition of temporal attribute measurement, is expensive enough to stop the drone from flying at 522.11% runtime overhead.

To tackle this challenge, we build on top of the observation that attestation on non-critical functionality of the system, such as tone processing of video, may not be necessary, thereby significantly reducing the number of events to be measured. However, when only a subset of code is attested, it becomes impossible to assess the security property of the execution, since security violations in the subsets that are not measured can impact the subsets that are measured. For example, an attacker can hijack the control flow in the tone processing library to call the release package command in mid-flight on a delivery drone. To enable the selective focus of attestation, we propose to leverage software compartmentalization to isolate different functionalities of the system, such that selective measurement of information flow events can offer the evidence for mission verification. Under this design, attestation events (control flow or data flow) can be measured at the inter and/or intra-compartment levels.

While selective attestation via software compartmentalization offers an opportunity to trade off the security of non-critical compartments for performance, existing system measurement techniques still impose prohibitive runtime overhead for real-time CPS. To minimize the timing impact of attestation on the real-time responsiveness of the system, we developed a set of techniques to reduce runtime overhead. To avoid context switching when storing the attestation events, we use the software fault isolation (SFI) technique to sandbox each compartment such that metadata can be stored directly within the application memory space without the need to trap into privilege code. To minimize the impact from temporal measurement, we proposed time recovery via re-execution where only the start and end time of critical compartment execution is recorded, and the temporal behavior within the compartment is recovered using the recorded control flow. Lastly, to minimize the logging impact on the real-time properties, we leveraged a ring buffer to enable the recording of the logs asynchronously. Furthermore, an independent real-time task is used to perform batch processing, and this task has to be taken into consideration in the schedulability analysis.

Since the high-level operational semantics and the desired level of protection for the compartments are often different for each platform and mission, the proposed approach has to be flexible enough to allow platform-mission-specific customization. ARI adapts a policy-based approach to enable customizable selective attestation of mission integrity. Using the user-defined policy, ARI automatically partitions the software and selects different levels of measurement granularity for individual compartments and their interactions.

We implemented ARI on both ARM Cortex-A and Cortex-M platforms. Using three concrete attack cases (on timing, control, and control flow), we show how ARI can be used for attestation of RMEI. We evaluated the performance on four real-time CPS applications, including drone, rover, syringe pump, and oxygen concentrator, over 20 different policies. For copters powered by ArduPilot, our system incurs only 10.7% runtime overhead when the safety-critical attitude controller is placed in the critical compartments, and no real-time tasks exceed their deadlines. In summary, we make the following contributions:

- We formulate a new security property, real-time mission execution integrity, that attests the integrity and timeliness of an autonomous CPS over the mission execution.
- We design and implement ARI, a policy-guided system to automatically compartmentalize and instrument the CPS software to provide measurements and verification for real-time mission execution integrity.
- We show three concrete use cases and evaluate ARI on four real-world embedded programs on two platforms under 20 different policies to demonstrate its performance and practicality. We also conducted a user study involving 21 developers of different skill levels to understand the usability.

2 Background and Motivation

2.1 Real-time System Security

Real-time Systems: For many systems that interact with the physical world, timing is of great importance to security and safety because the physical world clock continues to elapse. Considering the collision avoidance system on autonomous vehicles, a correct but untimely result from the proximity calculation does not offer much utility. Contrary to the popular

belief that real-time tasks have to be completed as quickly as possible, it is more important that real-time tasks are completed before the deadline. Real-time systems are often categorized into soft real-time and hard real-time systems [21]. In soft real-time systems, deadline misses should be minimized, but occasional misses are acceptable. In hard real-time systems, tasks cannot exceed the deadlines.

Software Security: Software instrumentation is a common technique to insert in-line reference monitors into the software. Compiler passes are used to insert assemblies into the program to enforce a specific security policy, such as control flow integrity (CFI) or data flow integrity. Software instrumentation can also be used for software fault isolation (SFI) [22] by sanitizing the destination of data access and control flow transfer instructions. More background is available in several seminal systematization of knowledge (SoK) papers [23, 24].

Timing Attacks: Besides conventional memory corruption attacks, real-time systems are also vulnerable to timing manipulation. Existing attacks that target system timing behavior often focus on either increasing task latency to cause deadline misses [17, 18, 25] or increasing task jitter to destabilize the control system [19, 26]. In a latency attack [17, 18], an attacker often leverages resource contention (such as cache pollution) to cause a significant delay in the victim task. In a jitter attack, an attacker aims to manipulate the timing of the system to cause jitter in the completion time of a task. Even though not all jitters are harmful, jitters in control actuation can lead to destabilization of the system [26]. In Butterfly attack [19], the attacker manipulates the GPS signal to influence the completion time of the actuation task, which causes the system to fail due to the control jitter. Existing defenses against attacks on jitter often involve injecting dummy tasks to prevent jitter in the task execution [27] or utilizing multiple controllers in the system [28]. However, they cannot detect or defend against control jitters introduced by compromised CPS software. ARI complements the existing defense systems by providing a mechanism to attest on the temporal properties and uncover attacks, even if the software is compromised.

2.2 Why Real-time Mission Integrity?

Recognizing the need to verify the integrity of externally-triggered actions in IoT [5] and CPS [4], recent efforts on remote attestation often focus on proof of correct execution that can detect control-flow and data-flow-related violations [3–5]. However, though such interaction paradigm is common for IoT, autonomous real-time CPS presents unique challenges.

Continuous Measurement on the Mission (Operations) Integrity is Important: Using a delivery done running ArduCopter [29] as an example, the request to re-route the drone to fly to a different waypoint to pick up an additional package will finish within several function calls, involving only the insertion of several waypoints. However, the exploitation of

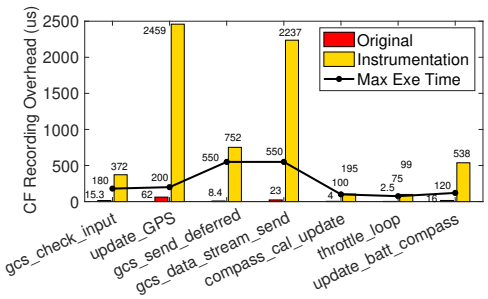


Figure 1: Control Flow Recording Overhead

waypoint controller via malicious waypoint data can happen much later in the mission. Furthermore, previous works [30] have also considered peripheral manipulation as a potential attack vector during the mission. Since autonomous systems have to defend against various attacks during the entirety of the mission to guarantee security and safety, it is important to consider the mission as the integration of all autonomous actions and examine its integrity with continuous measurement throughout the mission.

Temporal Property is also Important: Timeliness is reflected in ARI from two aspects. First, temporal behaviors need to be attested. The temporal measurement of the system has to capture not only the task completion latency (to ensure no deadline misses) but also jitter (to ensure there is no destabilizing control jitter). Second, the measurement process for attestation needs to have minimized and predictable impact on the temporal behavior of the system. All real-time tasks should still meet their deadlines. To understand the extent of the impact, we measured the overhead from direct application of existing approach for program behavior attestation. As shown in Fig. 1, when control-flow (CF) attestation is directly adapted to measure ArduCopter, many real-time tasks exceed their deadlines. A key factor contributing to such prohibitive overhead is the high frequency control loop for reading from sensors and writing to actuators continuously throughout the mission. For example, *fastloop* in ArduPilot runs at 400Hz.

2.3 Real-time Mission Execution Integrity

Intuitively, real-time mission execution integrity, RMEI, implies that the autonomous system shall execute the mission without deviating from the expected behavior (control/data flow behavior and temporal behavior). Different from existing literature on program execution attestation [4, 5] where there is a concise definition for the security properties such as control-flow (CF) and data-flow (DF) integrity, the measurement of RMEI has to make the trade-off between the level of details in the measurement and an acceptable level of runtime overhead for the real-time CPS. To enable this trade-off, the CPS software is decomposed into multiple loosely coupled logical compartments and only the information flow

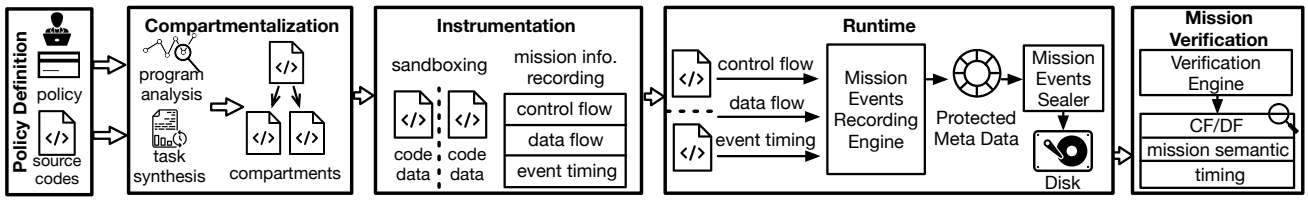


Figure 2: Overview of ARI

events between these compartments and within each critical compartment are recorded for verification. To ensure that compartments cannot tamper with each other, they are isolated using either memory controller or software instrumentation. By default, ARI measures the following mission execution attributes, 1) control flow in critical compartments and data flow on critical variables, 2) transfers among the compartments, and 3) timestamp of each compartment transfer.

A key observation is that every mission has its own unique requirement on how the system should be compartmentalized and what critical compartments or control variables should be measured. Inspired by recent developments in policy-driven compartmentalization techniques [31, 32], we propose to decouple the attestation policy from the actual implementation of the mechanisms. Based on the policy, ARI automatically compartmentalizes the software and inserts the instrumentation for measurement. Using the measurements, a verifier can then determine if there is any violation of RMEI.

3 Threat Model

Following the threat model in existing remote attestation work [3–5, 33, 34], we make the assumption that instrumentations cannot be bypassed or modified. We further assume that the trust anchor and clock are secure. The anchor can be ARM TrustZone [35] or RISC-V keystone [36], or the dedicated hardware design such as VRASED [37] or TrustLite [38]. As a result, our TCB includes instrumentation, the system that protects the instrumentation (such as the privileged code that operates MMU or MPU [31]), trust anchor, and the hardware. In practice, the code in the TCB that enables ARI, such as time measurement and measurement signing, should be attested as well. We assume the attacker can exploit vulnerabilities inside the CPS software stack to perform control flow hijacking or data-only attacks. He/she may also exploit weaknesses in system implementations to adversarially manipulate execution timing [19, 39–41]. Similar to other solutions that attest cyber-only behaviors [3–5, 33, 34], ARI is only able to detect program behavior violations on either program control (CF/DF) or temporal characteristic (delay/jitter). Leveraging the ability to record critical control variables and the timing profile, ARI can also be used to improve the possibility of detecting sensor attack [42] by recognizing the inconsistency in cyber domain with existing methods [43], but is unable to confirm the attack nevertheless.

4 ARI Design

Design Overview: ARI is a policy-guided real-time mission execution integrity attestation system. It builds on the intuition that by compartmentalizing the CPS application and measuring the high-level information flow between the compartments, it is possible to strike the balance between the fidelity of the program behavior and real-time performance.

Policy in ARI can be used to specify: 1) How to compartmentalize the software, 2) the criticality of functions and variables, where critical functions, and variables, as well as their dependencies, are always measured by default, 3) the type of program events to measure for each critical compartment, including control flow events, value-based data flow protection, execution timing, etc., and 4) the type of program events to measure between each pair of compartments. Depending on the level of customization desired by the user, ARI provides three ways of specifying the policy. Specifically, experienced users, such as software developers, can specify every option for each compartment. Users who are not familiar with the target software can use the built-in policies for compartmentalization and critical functions/variables annotation. Additional details on the API are available in Appendix B.1.

Given a policy, the high-level workflow of ARI is shown in Fig. 2. There are four main steps. At compile time, the program is first partitioned into different compartments using the source code and compartment policy. Instrumentation for compartment sandboxing and event measurement is then added using a customized compiler based on LLVM. At runtime, the system states are measured using these instrumentations. The measured events are then signed and stored using a runtime library in the TCB. When the logs are collected (often at the end of the mission if network connectivity is limited), the verification engine then verifies the mission integrity.

Key Challenges and Approaches: There are two key challenges in the design of ARI, from the perspectives of scalability (due to mission-level attestation) and real-time performance overhead (due to security measurements). Scalability is addressed by measuring only events inside critical compartments and at the inter-compartment level. Compartmentalization techniques are used to prevent the unmeasured non-critical compartments from impacting the safety-critical controls in the critical compartments. Even with the reduction in the number of measurable events, the impact on real-time performance is still significant. To address this challenge, ARI

leverages a set of techniques including customized encoding, verification by re-execution, instruction-based time estimate, and batch processing to meet the real-time constraints.

4.1 From Policy to Compartments

Compartmentalization is essential to the design of ARI. Its ability to prevent vulnerability in one compartment from influencing another is the foundation for the trade-off mechanism in ARI. It allows the system designer to have different levels of granularity in program behavior measurement across different compartments, empowering the design space exploration on the trade-off between performance and security.

A compartment is defined as an isolated code and data region. Each function belongs to exactly one compartment. Different from previous efforts [31, 32] that attempt to maximize the security protection based on the principle of least privilege, there is an additional dimension of consideration for real-time performance overhead in ARI. Smaller compartments can lead to prohibitive performance overhead for the real-time system, even though it may provide finer-grained program behavior measurements for attestation. As a result, it is important that the program is carefully partitioned.

Policy Generation: Similar to the existing software compartmentalization approach, a policy empowers system designers to build customized protection for the target while abstracting the complexity of the protection mechanisms [32, 44]. However, defining a policy requires careful consideration of the concrete threats to the system under protection. Existing autonomous CPSs often operate in diverse environments facing unique threats in different missions. To facilitate a more effective generation of policies for the system designers, ARI provides different levels of customization through various APIs. Expert users can write a script to fully specify how compartments should be generated, their criticality, as well as the granularity of measurement on the program behaviors. Alternatively, users can also directly use the built-in partition and criticality annotation policies. File-name-based or module-based built-in partition policy places functions under the same file or directory in the same compartment. Controller-based or peripheral-based partition policy gathers codes of a controller or a peripheral operation into a compartment. With a built-in partition policy, ARI can also automatically merge compartments that have the most frequent control flow transfers if users choose to specify the number of compartments to reduce inter-compartment communication. Controller-based, actuator-based, and sensor-based built-in annotation policies label the functions related to controllers, actuators, and sensors as critical ones. Variables used in critical functions are also annotated as critical by default.

Compartment Generation and Labeling: ARI automatically partitions the software into different compartments and then labels their criticality. Based on the given policy, ARI first partitions the program dependency graph (PDG) into subgraphs. The basic blocks that form the structure of the compartments are then grouped into the same logical section with their local data dependencies, to which an isolation mechanism can be added. Each compartment that has functions or variables annotated as mission-critical/safety-critical, either manually by the developer or automatically by ARI, is labeled as critical. By default, inter-compartment events as well as critical intra-compartment events are measured and signed.

4.2 Compartment Isolation

Since program execution is not measured in non-critical compartments due to the real-time constraint, it is crucial to enforce strong isolation between measured (critical/secure) and unmeasured (non-critical/non-secure) compartments for the attestation to be meaningful. Existing approaches towards isolation generally rely on either hardware security features [31, 32] or privilege separation [44]. However, adopting these approaches to autonomous CPS presents two new challenges. First, hardware platforms on CPS are diverse. Some hardware features used in existing work may not be available. Second, existing works often store metadata in higher privilege memory (either privilege/kernel mode or TEE). This can be too expensive for real-time CPS due to costly context switches. To tackle these challenges, software fault isolation (SFI) [22] is used to enforce compartmentalization in ARI. This enables metadata storage without context switches, minimizing the impact on the real-time property. Furthermore, it can be generalized to different hardware platforms including ARM Cortex-A and Cortex-M. There are three key isolation efforts, control flow, data flow, and stack respectively.

Control Flow Isolation: In ARI, control flows have two additional dimensions. Besides forward or backward, control flows can also be intra-compartment or inter-compartment, and the compartments can be critical/secure or non-critical/non-secure. Since program behaviors are not attested inside non-critical compartments for performance, all control transfers (indirect/direct) from non-critical compartments to critical compartments are measured, such that control flows can be attested in critical compartments. Depending on the granularity of mission integrity, control flows between non-critical compartments can also be recorded. The idea is that compartment-level transfers can provide additional mission semantics. The most direct design is to add instrumentation at each indirect branching instruction to distinguish if it is an inter-compartment transfer or an intra-compartment transfer. However, this significantly increases the performance overhead. After analyzing the root cause, we found that many function returns are intra-compartment call returns and can be directly sandboxed. However, they can also be invoked externally. To reduce the return indirect branching checking, we create stub wrappers for the external invocations to avoid destination checking on function returns.

Data Flow Isolation: Data flow isolation is motivated by several requirements in ARI. First, metadata has to be protected. However, storing them in the secure world or privileged memory introduces prohibitive runtime overhead CPSs. As a result, all memory access instructions have to be sandboxed to enable secure storage. Second, critical variables also have to be protected, we adapt the value-based data flow protection technique [5] to enable this protection. Lastly, the sandbox can also harden the system by enforcing isolation according to the principle of least privilege among different compartments. Following existing SFI mechanisms [22, 45], ARI makes use of a mask register to implement the mechanism. However, different from conventional SFI that aims to sandbox a single piece of code, often without the need to access the external memory, ARI has to use address masking to create multiple compartments while facilitating access to shared memory. The existing approach to accomplish this is to reserve more general-purpose registers for masking [46]. To avoid this, we leverage static analysis to recognize instructions with memory access destination outside the compartment and add instructions to update the mask before the access. Since the number of such instructions is much smaller, the performance overhead is substantially smaller than monopolizing additional general-purpose registers for masking. The trade-off is that attacker will have one more gadget for memory tampering, however, all access remains confined within the sandbox.

Stack Isolation: ARI needs to isolate stack between different compartments. To efficiently accomplish this, a separate stack is allocated for each compartment adjacent to the compartment's code in memory. Such design enables control flow and stack isolation using a single reserved register. There are two challenges when isolating compartments' stacks using SFI. First, local variable accesses are frequent in CPS, (e.g., 50% write instructions using stack pointer (SP) relative addressing in ArduCopter). Thus, masking every SP address still incurs significant overhead. To minimize the performance overhead, only SP-changing instructions are masked. To prevent attackers from accessing memory outside the sandbox using SP relative addressing instructions, ARI puts redzone around each stack. The size of the redzone in each compartment is the maximum relative address in SP relative addressing instructions. However, the relative address can be large for some compartments. This significantly increases the memory overhead due to the insertion of redzones. To further narrow down the memory overhead, a subset of relative addressing instructions is also masked to avoid unnecessarily increasing the size of the redzone. Second, the sandbox prevents memory access across stacks. ARI solves this problem by replacing cross-stack memory accesses with instrumented trampolines.

4.3 Real-time Attested Event Measurement

Besides information flows, temporal behavior also has to be measured, while meeting the real-time constraints.

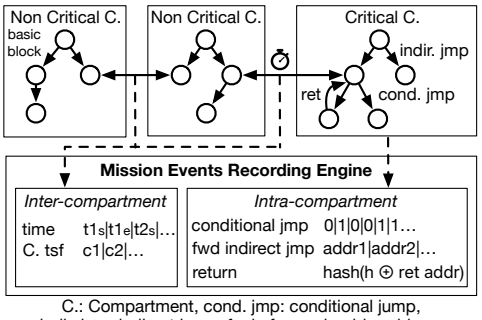


Figure 3: Inter/Intra-Compartment Events Recording

Measurement of Inter/Intra Compartment Flows: Embedded CPS can execute a complex control loop that produces a significant amount of CF events (e.g., 3.1M CF events per second on ArduCopter). Therefore, recording every CF event can lead to a prohibitive runtime overhead. To address this challenge, only the inter-compartment CF events and critical intra-compartment CF events are recorded by default. For an inter-compartment CF event, ARI measures the source and destination of the event. To measure an intra-compartment-level CF event, conditional jump decisions, forward indirect jump targets, and the hash of return addresses are measured to allow for the reconstruction of CF, as shown in Fig. 3. To reduce measurement log size, all return addresses are hashed into one hash value and only this hash value is recorded.

Measurement DoS Prevention: When the control flow attestation requirement is relaxed in the non-critical compartments, it is possible for an attacker to overwhelm the measurement process by repetitively branching between two non-critical compartments to create a large number of events to be measured. While none of the existing work, including ARI, can prevent this type of DoS attack, ARI leverages the unique control loop design and predictability in CPS systems to mitigate the risk by imposing a maximum measurement log size on each control loop. Such maximum measurement log size can be conservatively estimated. This mitigation ensures that the mission log continues even if the system might be under attack for continuing attestation.

Measurement of Temporal Behaviors: Temporal property is critical to real-time cyber-physical systems, due to the unique requirement to perform timely sensing and actuation. A key aspect of RMEI is the attestation of the temporal characteristics of the CPS. As discussed in the timing attacks, the latest developments in this direction have evolved beyond simple DoS that causes real-time tasks to miss their deadlines [18, 19, 40, 41]. As a result, similar to how attestation on program behaviors needs to capture dynamic program behaviors beyond the checksum of code pages, attestation on program temporal behaviors needs to capture more fine-grained timing information that will allow the verifier to understand

other temporal attributes, such as control jitter [28].

To obtain instruction-level temporal properties, the function that consists of multiple instructions for clock reading and timestamp storage has to be added per instruction. However, this design not only adds prohibitive runtime delay to the system, causing a significant impact on the real-time property, but also introduces a large number of measurements that can easily overwhelm the real-time CPS. Inspired by the approach that conducts control flow attestation verification through re-execution [5], we propose *timing through re-execution* to tackle this challenge. More specifically, since all the control flow information is recorded in the critical compartments to allow for the reconstruction of execution traces for attestation, it is possible to re-estimate the execution timing of individual basic blocks via the re-execution as long as the time of entry and exit is recorded. Using the duration and control flow of the program, as well as a profile of instruction time cost, it is possible to attribute the time of execution to a much finer granularity like basic block and instruction. It is important to note that the recovery of time through control flow only provides an estimation, since both the inputs and architectural/system states can also impact timing, but are not recorded. Another challenge is the impact of interrupts. However, timing variation from interrupts is often taken into consideration for real-time systems. Unless there is a significant shift in the distribution, its ability to harm the system is limited [47].

Minimizing the Recording Overhead: There are two primary efforts in minimizing runtime overhead from recording. *1) Measurement Event Encoding:* A naive representation of a recorded event can be a pair of source and destination addresses. To reduce the log size, we encode the event to eliminate redundant information. *2) Using SFI to securely record without context switching:* Existing approaches generally record measurements synchronously and store them in the memory space of privileged software [3, 5]. ARI leverages the isolation provided by the SFI to store the measured events securely in the same address space using a lock-free structure, which is then transferred to the TCB for signature. More details are available in Appendix B.3.

Understanding the Real-time Impact of ARI: It is important to ensure the attestation system does not cause excessive run-time overhead leading to the violation of real-time guarantee. For hard real-time systems, tasks have strict deadlines. In order to determine if a performance overhead is acceptable, schedulability analysis using the worst-case execution time (WCET) of each task is used to ensure that the system will remain schedulable after the software instrumentation is added for the attestation. To enable the schedulability analysis, ARI provides analysis on the runtime overhead due to attestation by estimating the WCET of a task based on the worst case execution path (WCEP) using tools such as aiT [48]. The developer can then use the new post-instrumentation WCET to determine if the system can tolerate the overhead using schedulability analysis [49]. For all of the policies in our evaluation, the system remains schedulable. For a soft real-time system, ARI calculates the average runtime overhead aggregated over the expected missions. It is also common for CPS developers to experimentally profile the program in the target environment, which is shown in our evaluation.

4.4 Verification on the Measurement

By default, ARI verifies inter-compartment control transfers, intra-critical-compartment indirect jumps, policy-specified variable values, and critical compartment execution timing during the verification phase, as shown in Fig. 4. In order to balance security and real-time performance, users may select a subset of the above mission events to verify. For both the inter-compartment and the intra-compartment forward jumps, the target is checked against the offline generated CFG. Even though the inter-compartment forward direct transfer can only branch to valid deterministic targets, measuring the inter-compartment forward direct jump is still useful since control flow details inside the non-critical compartments are not measured. The integrity of the inter-compartment backward return address is checked against the measured inter-compartment forward jump to ensure an inter-compartment return can only return to the compartment where it jumps from. The integrity of intra-compartment returns is verified in a different manner. Specifically, ARI compares the recorded hash value with the reconstructed hash value of intra-compartment return addresses. To reconstruct the correct intra-critical-compartment return address hash, ARI emulates mission execution on application binary from the mission entry point. The execution path of a critical compartment in the executed mission is reconstructed with the help of control flow measurement logs.

To verify the modified value of policy-specified critical values and timing, the verification engine checks whether the recorded value and timing satisfy the expected mission-specific patterns. More specifically, the execution profile of each basic block in the critical compartments is compared against the measured record, the verification passes only if the measured execution time falls in the execution profile. While the execution profiles can be mission-specific, common functions such as updates on Kalman Filter [50], that provides estimates of some unknown variables based on history, often have very small variations. It is important to note that even though it is possible to detect deviation from expected behavior in the software system, an attestation system often cannot provide root cause analysis or attribute the attack. It has to be used in conjunction with tools, such as Mayday [51].

5 Implementation

ARI implementation includes three parts, application compartment generation, program instrumentation, and mission integrity verification engine. More details are in Appendix B.

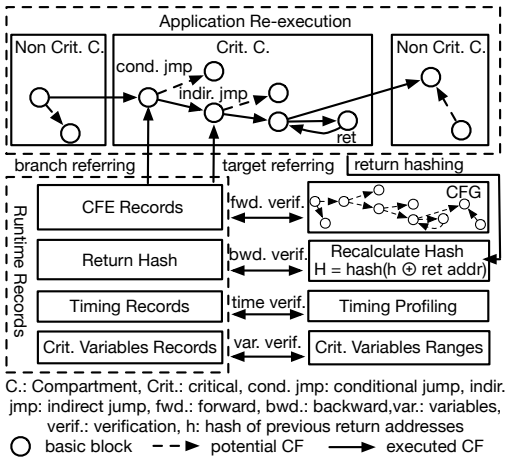


Figure 4: Verification on Measurement

Compartment Generation: Two LLVM passes and a Python script are implemented for automatic compartment generation. The first front-end LLVM pass is for the generation of the PDG in policy definition. The second front-end LLVM pass is used to group functions and variables of the same logical compartment into the same ELF section. Lastly, the compiled object files are linked together using the linker script. Due to the masking technique for SFI, both the starting address and the size of the compartments are aligned to a power of 2 .

Program Instrumentation: A front-end LLVM pass and a back-end ARM LLVM pass are used for program instrumentation. The front-end LLVM pass is used to identify and label the locations of compartment transfers, global data accesses, and cross-stack memory accesses. While the back-end ARM LLVM pass is used to insert compartment transfer trampolines, control flow recording, critical data flow checking, timing recording functionalities, and inlined address masking used in SFI. The measurement is sealed and saved using trusted application supported by OP-TEE [52].

Verification Engine: The verification engine is implemented based on Capstone [53]. The assembly is traversed to emulate mission re-execution. Forward control flow and temporal measurement are attested during re-execution. Backward control flow is attested by comparing return hash in the measurement and the reconstructed one. Critical variables are verified through comparing expected value to recording from critical variable instrumentation.

Difference between Cortex-A and Cortex-M: Our prototype is implemented on both Cortex-A53 and Cortex-M33. Since Cortex-A and Cortex-M have different instruction sets and system service supports, the differences in the implementations mainly include back-end pass implementation, timestamp-getting methods, and TEE supports. For example, the timestamp is obtained by invoking system calls on Cortex-A53 and by directly reading MMIO registers on Cortex-M33. For TEE supports, ARI uses OP-TEE as the secure OS on Cortex-A and the vendor provided OS for Cortex-M.

6 Case Study on Autonomous Drone

We conducted three case studies on drone to showcase how ARI can attest mission integrity without causing deadline misses in real-time tasks. In this section, the discussion will focus on the timing attack. Additional cases on control and software can be found in Appendix A.

Butterfly Attack - Temporal Violation: Minimizing control jitter is important for CPS [28]. Butterfly attack [19] adversarially manipulates the control task's scheduling jitters to destabilize the target CPS. Concretely, an adversary may manipulate inputs to reduce the computation of a non-critical task to cause the completion time of critical control tasks to be moved forward. By alternating between early termination and regular execution of the non-critical task, the attacker can cause jitters in the completion time of control task, thus the actuation. When the actuation jitters exceed the tolerable range, the system will start to destabilize.

Attack implementation: We follow the same configuration in [19], and implemented the butterfly attack on ArduCopter, where an adversary jams the GPS signals such that a non-critical task that updates GPS data will be skipped. We simulate GPS signal jamming by setting the value of *gps_updated* flag which indicates the availability of GPS messages. By setting this variable to *false*, the program will skip the function *camera.update()*, which leads the task *update_GPS()* to finish earlier. The earlier termination of *update_GPS()* leads to early invocation of *run_nav_updates()*, which is responsible for actuating command. Following [19], every three out of four GPS messages are jammed.

Detection implementation: To detect timing anomaly, we deploy ARI using controller-based compartmentalization with navigation feature in ArduCopter labeled as a critical. With the recorded entry and exit time of *run_nav_updates()*, there is a clear evidence of jitters from the invocation time, indicating a possible attack. There is, however, a 7.43% runtime overhead and a 4.27% memory overhead. Though there are other detection mechanisms [54, 55], they make the assumption of trusted software. ARI is robust against the combination of butterfly attack and memory corruption.

7 Evaluation

In this section, we try to answer the following questions: (1) What is runtime and memory overhead of ARI? (2) How scalable is ARI? (3) How much manual effort is required to use ARI? To answer the above questions, (1) we measure runtime and memory overhead under different policies on both Cortex-A and Cortex-M platforms; (2) we show real-time task runtime and memory overhead when the policy specifies different numbers of compartments and critical compartments; and (3) we conduct a user study to understand the level of manual efforts required.

Experiment Setup: To demonstrate the generalizability of ARI, we choose two relatively complex and widely used autonomous systems, i.e., ArduCopter (AC) [20], ArduRover (AR) [20], and three small code base embedded systems, i.e., House Alarm (HA), Oxygen Concentrator (OC), and open Syringe Pump (SP) as test applications and evaluated them on both Cortex-A and Cortex-M platforms. The deadlines of real-time applications can be inferred from source codes. On Cortex-A platforms, ArduCopter and ArduRover run on Rpi3 with NAVIO2 daughter board in simulation. Syringe pump and house alarm run on Rpi3. We also run ArduCopter on a self-built real-life drone. On Cortex-M platform, we migrate the syringe pump and oxygen concentrator into FreeRTOS on NXP LPCXpresso55S69 development board. Our verification engine runs on a workstation with 32 GB memory and Intel Xeon W-2123 CPU.

7.1 System Overhead

Experiment Policies: To evaluate ARI performance overhead under different policies, we use four different policies in each application.

Compartmentalization Policy: Peripheral-based (peri.) or controller-based (cont.) policies groups functions based on the peripheral the code is operating on or the controller the code belongs to. We applied these two to ArduCopter and ArduRover, due to the complex control in these systems. File-based (file), operation-based (oper.), and module-based (modu.) compartmentalization policies group functions based on the file they belong to, the operations they support, and the module they are in respectively. These policies are applied to smaller applications including the syringe pump, house alarm, and oxygen concentrator, due to the relatively simple program structures in these applications.

Critical Compartments and Variables: Autonomous vehicles, such as ArduCopter and ArduRover, highly depend on attitude controllers and fail-safe controllers to function properly. Thus, for ArduCopter, we label the compartments that contain the fail-safe controller (fs) and attitude controller (at) as critical. For ArduRover, the fail-safe controller (fs) and the crash-check controller (cc) are marked as critical. CPS often comprises sensing, actuating (I/O), and controlling (command decision) phases. Thus, the command decision and I/O compartments are annotated as critical ones for syringe pump, house alarm, and oxygen concentrator. Specifically, the bolus (bl) and serial (sr) compartments are marked as critical for syringe pump, as they perform injection and serial processing. Similarly, I/O control and command (cmd) execution are marked as critical for house alarm. Valve control (vc) and serial processing (sr) are marked as critical for oxygen concentrators. Furthermore, all control variables and I/O operation variables are also annotated as critical.

Missions: To evaluate system overhead generated by ARI, we execute different missions on different platforms as follows. On Cortex-A platform, we run ArduCopter and ArduRover with CMAC-circuit (the built-in circle cruising test) mission trajectory for 1 hour, and syringe pump 10000 times. On Cortex-M platform, we run oxygen concentrator for 1 hour. The syringe pump and house alarm are executed 10000 times. The real-time tasks and real-time parameters, such as the deadline, are extracted from application source codes.

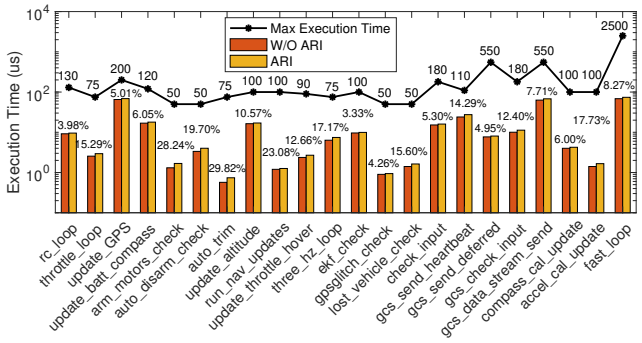


Figure 5: Tasks Execution Runtime Overhead in ArduCopter (controller-based policy with attitude controller as critical)

Run-time Overhead: Using average task execution time, we aim to understand the overhead of ARI on the CPS on ArduCopter, ArduRover, syringe pump, and oxygen concentrator. For house alarm the operation execution time is used due to its operation-based design. As shown in Tab. 1, the highest runtime overhead on Cortex-A and Cortex-M platform are 17.2% (on ArduCopter) and 15.8% (on syringe pump) respectively. Generally, the policy that attests more complex critical compartment or with more compartments incurs higher runtime overhead.

Real-time Performance Impact: To evaluate the real-time performance, we measure the deadline miss rates and conduct the schedulability analysis.

Real-time Deadline Miss for Soft Real-time Systems: We measure each real-time task execution time by dynamically running the same 1-hour mission 10 times or operation 10000 times, and check whether the average task execution time exceeds its maximum execution time. We measure the execution time of 22 ArduCopter real-time tasks before and after deploying ARI on a real-life quadcopter. To show the worst case under our tested policies, we choose control-based compartment policy with attitude controller marked as critical, which has the highest runtime overhead in Tab. 1. As shown in Fig. 5, the runtime overhead of each task ranges from 3.98% to 29.82% and none of the 22 tasks exceed its maximal execution time. In addition to ArduCopter, we run ArduRover, syringe pump on Pi3, syringe pump, oxygen concentrator on NXP LPCXpresso55S69 development board. None of real-time tasks in ArduRover, syringe pump, and oxygen concentrator miss their deadlines. More details can be found in Appendix C.

Table 1: Application Manual Effort Statistics and Runtime Overhead ARI

CPS	Policy		#Cpt.	Manual Effort Statistics			System Overhead Statistics			
	Cpt.	Crit.		scr. size	func.	vari.	Verification	Log Size ↓	Execution Time ↑	Memory Size (MB) ↑
ArduCopter(AC)	cont.	fs	12	36	4	5	1.35 min	0.32/1.16 GB (27.3%)	79.3/69.6 ms (13.9%)	4.31/4.13 (4.36%)
		at	12		11	18	10.37 min	0.65/1.16 GB (56.2%)	81.6/69.6 ms (17.2%)	4.31/4.13 (4.36%)
	peri.	fs	8	78	4	5	1.22 min	0.03/1.16 GB (2.2%)	74.4/69.6 ms (6.7%)	4.30/4.13 (4.12%)
		at	8		11	18	8.88 min	0.39/1.16 GB (33.4%)	77.6/69.6 ms (10.7%)	4.30/4.13 (4.12%)
ArduRover(AR)	cont.	fs	11	36	4	2	1.08 min	4.82/999 MB (0.4%)	23.2/20.1 ms (15.4%)	4.01/3.87 (3.62%)
		cc	11		4	4	9.33 min	1.45/999 MB (0.1%)	22.6/20.1 ms (12.4%)	4.01/3.87 (3.62%)
	peri.	fs	6	78	4	2	1.16 min	76.5/999 MB (7.7%)	22.8/20.1 ms (13.4%)	3.99/3.87 (3.10%)
		cc	6		4	4	7.64 min	80.6/999 MB (8.0%)	22.0/20.1 ms (9.5%)	3.99/3.87 (3.10%)
Syringe Pump(SP/A)	file	bl	4	61	2	6	1.7 s	11/12 B (91.7%)	119/109 ms (9.2%)	0.037/0.034 (8.82%)
		sr	4		4	9	0.8 s	2/12 B (16.7%)	117/109 ms (7.3%)	0.037/0.034 (8.82%)
	oper.	bl	3	46	2	6	1.7 s	10/12 B (83.3%)	113/109 ms (3.7%)	0.036/0.034 (5.88%)
		sr	3		4	9	0.8 s	2/12 B (6.5%)	112/109 ms (2.8%)	0.036/0.034 (5.88%)
House Alarm(HA)	file	I/O	8	61	6	2	0.51 s	16/17 B (94.1%)	2.067/2.066 s (0.04%)	0.029/0.027 (7.41%)
		cmd	8		9	3	0.53 s	5/17 B (29.4%)	2.068/2.066 s (0.09%)	0.029/0.027 (7.41%)
	oper.	I/O	6	46	6	2	0.45 s	9/17 B (52.9%)	2.067/2.066 s (0.04%)	0.029/0.027 (7.41%)
		cmd	6		9	3	0.49 s	7/17 B (41.2%)	2.068/2.066 s (0.09%)	0.029/0.027 (7.41%)
Syringe Pump(SP/M)	file	bl	4	61	4	6	1.7 s	11/12 B (91.7%)	132/114 ms (15.8%)	0.036/0.032 (12.5%)
		sr	4		8	9	0.8 s	2/12 B (16.7%)	127/114 ms (13.1%)	0.036/0.032 (12.5%)
	oper.	bl	3	46	4	6	1.7 s	10/12 B (93.3%)	129/114 ms (13.2%)	0.035/0.032 (9.4%)
		sr	3		8	9	0.8 s	2/12 B (16.7%)	121/114 ms (6.1%)	0.036/0.032 (12.5%)
Oxygen Concent(OC)	file	vc	13	61	3	6	10.0 s	1.68/10.4 MB (16.2%)	1.949/1.873 ms (4.06%)	0.358/0.325 (10.2%)
		psa	13		5	7	10.8 s	1.74/10.4 MB (16.7%)	1.937/1.873 ms (3.42%)	0.359/0.325 (10.5%)
	modu.	vc	7	46	3	6	9.6 s	1.53/10.4 MB (1.47%)	1.931/1.873 ms (3.10%)	0.355/0.325 (9.2%)
		psa	7		5	7	10.7 s	1.64/10.4 MB (1.58%)	1.908/1.873 ms (1.87%)	0.355/0.325 (9.2%)

cont.(controller), peri.(peripheral), file(file name), oper.(operation), modu.(module), fs(fail safe), at(attitude), cc(crash checker), bl(bolus), sr(serial), cmd(command), vc(valve control), Cpt.(Compartment), Crit.(Critical Compartment), var.(variables), func.(functions), scr.(script). ↑ means percentage of increment. ↓ means percentage of decrement.

Table 2: Schedulability Analysis Result

CPS	Task	WCET	WCET*	Period	Alg.	BW	BW*
SP	Inj	247 ms	277 ms	2000 ms	EDF	14.5%	16.0%
	SL	72 us	N/A	1583 ms			
OC	PSA	11.1 us	11.2 us	100 ms	RM	27.0%	32.5%
	SR	543 us	545 us	25 ms			
	DO	2541 us	2569 us	100 ms			
	SL	72 us	N/A	137 us			

Inj(Medicine Injection), SL(Measurement Sealer), PSA(Pressure Swing Adsorption), DO(Device Operation), SR(Sensor Read), BW(CPU Bandwidth), Alg.(Scheduling Algorithm), EDF(Earliest Deadline First), RM(Rate Monotonic), * with ARI

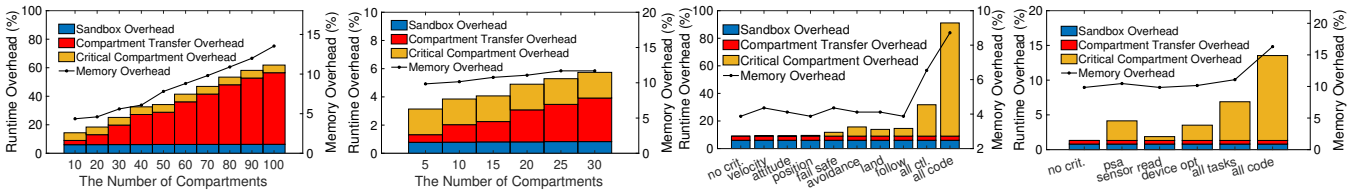
Schedulability Analysis for Hard Real-time Systems: To understand the real-time impact of ARI on hard real-time system, we analyze the schedulability with the additional security provided by ARI. More specifically, the new WCET time is obtained using aiT WCET Analyzers [48], an industry-level WCET tool for two hard real-time applications, i.e., syringe pump and oxygen concentrator. We then use CARTS [49] which is a compositional scheduling analysis tool with task WCET, deadline, and the scheduling algorithm from the software as inputs. As shown in Tab. 2, both syringe pump and oxygen concentrator are schedulable with ARI.

Memory Overhead: We measure memory usage, i.e., RAM usage size on Cortex-A, sum of RAM and Flash usage size on Cortex-M, of different applications. As shown in Tab. 1, ARI increases memory size by at most 8.82% and 12.5% on Cortex-A and Cortex-M platforms respectively. The memory size overhead depends on the policy. More compartments generally introduce more overhead because of (1) sandbox isolation requirement on compartment alignment, (2) additional stub wrappers, and (3) more stack redzones. Compartment alignment on cortex-M introduces more memory overhead than cortex-A because of a lack of virtual memory. The overhead of ARI is acceptable since the overhead is less than 1MB out of 1GB on Pi3 and 45KB out of 960KB on the microcontroller. To further understand the practicality, we survey the memory utilization of 10 CPS applications on low-end platforms [5, 31, 44] and measure 2 CPS applications on high-end platforms. On low-end platforms which often have 700KB to 1MB memory capacity [56, 57], CPS applications consumes 7KB to 125KB memory. The applications range from simple light controller to complex network applications, e.g., TCP-Echo. On high-end platforms which often have around 1GB memory [58], the runtime memory consumption of ArduCopter and ArduRover is less than 10 MB. Therefore, the corresponding memory overhead is acceptable.

Measurement Size and Verification Time: As shown in Tab. 1, compared with the solution which records all control flow events, ARI reduces the size of mission measurement logs to 34.7% on average under all evaluated applications and policies. ARI attributes this optimized result to not measuring non-critical compartments and optimized encoding. The verification of control flow integrity takes at most 10.37 minutes per hour mission time. It can be further reduced by parallelization. More details about system performance under different timing recording granularity can be found in Appendix C.

7.2 Scalability

Two important factors impact the scalability of policies: number of compartments and number of critical compartments. To understand the trade-off, we evaluated the performance



(a) #Compartments (Cortex-A) (b) #Compartments (Cortex-M) (c) #Critical Compartments(Cortex-A) (d) #Critical Compartments(Cortex-M)

Figure 6: Scalability Analysis

of ARI under different total numbers of compartments as well as different percentages of critical ones. ArduCopter on Cortex-A and oxygen concentrator on Cortex-M are used in this evaluation due to their highest system overhead.

Number of Compartments: To create policies with different numbers of compartments, we use a filename-based compartmentalization policy. Compartments are merged based on how frequently they communicate with each other to generate an arbitrary number of compartments. As shown in Fig. 6(a) and Fig. 6(b), both runtime and memory overhead increase as the number of compartments increases on both Cortex-A and Cortex-M. The increased runtime overhead is caused by the increased inter-compartment communications (i.e., compartment switching and timing recording overhead) and data accesses (i.e., shared data and cross-stack data accesses). The memory overhead is mainly caused by inter-compartment communication trampolines, stack redzones, and stub wrappers. Lastly, tasks in Copter are able to meet all deadlines when the number of compartments is less than 80.

Number of Critical Compartments: To evaluate system overhead under different numbers of critical compartments, we randomly select compartments derived from policy-specified compartmentalization as critical ones. We choose controller-based and filename-based compartmentalization policies for ArduCopter and oxygen concentrator since they have higher runtime overhead. As shown in Fig. 6(c) and Fig. 6(d), the runtime and memory overhead increase as the number of critical compartments increases. Generally, an increasing number of critical compartments generates more control flow events that need to be recorded. Since program information flow events can vary significantly among different compartments, the selection of critical compartments can have more impacts on runtime and memory overhead.

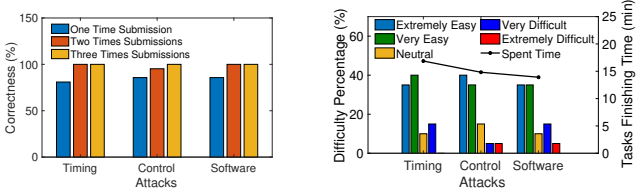
7.3 User Study

Our user study aims to evaluate two aspects of ARI: usability and extensibility. The usability study examined the required manual efforts of using the system to detect a specific known attack, while the extensibility study evaluated whether it is easy for users to customize solutions by building on top of built-in capabilities. We measured extensibility by asking the participants to optimize their initial policy from the usability study to improve system performance. We conducted the user studies both qualitatively and quantitatively, surveying participants' user experience and measuring policy accuracy, time spent on policy implementation, and script size. Our user study is approved by our university's IRB.

Recruitment: Our user study was conducted with known contacts to ensure diversity in the development experience, with a total of 21 participants. Participants have diverse development experiences (1-5 years) and backgrounds (from industry (6) to academia (15), from CPS (9) to security area (18)). About 24% of the participants have no prior exposure to TEE or SFI.

Procedure: All participants were asked to fill out their demographic information at the beginning. They are then provided with a tutorial that explains the working principles, APIs offered by ARI, as well as a concrete example of how the system can be used to detect a buffer overflow attack. Participants were asked to finish two tasks and keep track of the time they spent, including reading the documentation and optimizing policies. The first task is to detect three types of attacks (timing attack, control attack, and software attack) by choosing a default built-in compartmentalization and annotation policies from ARI, which were discussed in Section 4.1. The second task is to customize the aforementioned policies with lower runtime overhead. We consider a policy to be correct if it can be used by ARI to detect the attack successfully. The only feedback participants would receive during the process is whether their policy leads to successful detection. Participants were asked about the time spent and their experience. We tried to minimize the impact of desirability bias by communicating with the participant that the goal is to improve the design through the exercises.

Usability: We evaluated usability by collecting policy correctness, average time consumption, and difficulty score of defining a policy in each task. Participants will rate their perceived usage difficulty on a 5-point Likert scale [59] at the end of the survey, with 1 representing extremely easy to use and 5 being extremely difficult to use. The final difficulty score is the average of all responses from the participants. The correctness of the policies in the first submission reached 93.7%. Participants took 7.3 minutes on average to choose a default policy and the average difficulty score was 1.85. With



(a) Correctness of User Policy (b) Difficulty of Using ARI

Figure 7: Usability of ARI with Customized Policy

further investigation, we found that most incorrect policies were used to detect timing attacks and control attacks. Incorrect policies are mostly generated by participants that did not have experience with robotic vehicles. However, all of them submitted the correct policy on their second attempt.

Extensibility: To understand extensibility, we evaluated the easiness of building a customized policy from the default. Among the submitted policies, at most 3 critical variables and 2 critical functions are annotated, with all scripts containing less than 56 lines of code. As shown in Fig. 7, the average difficulty score in Task 2 was 2.12. While the default policies generate 41.6% runtime and 10.11% memory overhead, participants were able to generate new customized policies with 10.56 % runtime overhead and 7.51% memory overhead. The performance improvement comes from avoiding unnecessary critical variable annotation and a larger number of critical compartments. These customizations (i.e., reducing 31.49% runtime overhead and 3.05% memory overhead on average) require additional 6.9 minutes from the participants. In addition, the average difficulty score is increased by 0.27, which shows extensibility with moderate additional manual effort.

8 Security Analysis

Compartment Isolation: Attackers may attempt to break out of the sandbox. However, all indirect jump and memory access instructions use the reserved general purpose register r_{rsv} to store the destination address. Furthermore, the high bits of r_{rsv}, which define the boundary of the compartment sandbox, are only updated by the reference monitor upon entering or exiting the compartment, or accessing global data. Therefore, skipping the mask instruction does not help the attacker. As a result, reference monitors that handle inter-compartment transfers are the only gateway to escape compartmentalization. However, these inter-compartment transfers are measured.

Timing Attack: Timing attacks manipulate the temporal property of the computation, such as introducing latency to make tasks miss deadlines or introducing jitters to destabilize the system. Therefore, by examining the timestamps of the measured events, it is possible to detect timing attacks. However, in order to meet the real-time performance of the system, not all the timing information can be recorded. Even though it is often possible to detect timing manipulation attacks using just the timestamps of the inter-compartment transfer, the correct configuration of the policy remains important for attack detection, similar to other policy-based approaches [31].

Security on the Timing Impact from ARI: ARI changes the temporal behavior of the CPS, which may open up new attack surface. From the latency perspective, an attacker may use a compromised non-critical compartment to run an infinite loop to exhaust system resource. However, this would only exhaust the budget of the non-critical tasks without impacting critical tasks' ability to meet their deadlines due to the real-time task model and scheduler. From the jitters perspective, runtime overhead introduced by ARI has the potential to be misused to introduce jitters. Upon instrumentation, system designer can examine the newly introduced attack surface to determine if the threat can be mitigated, since not all jitters are harmful. If this is indeed a concern, dummy instructions can be used to maintain the original temporal profile variance. More analysis is in Appendix D.

9 Related Work

CPS Attack and Defense: Besides software attacks, there has been significant interest in understanding attacks from the physical domain [42, 60, 61]. To detect CPS attacks, defenders turn to physics to recognize violations of system invariants [62–64], often using learning-based approaches [64] or rule-based approaches [65]. ARI supplements these works by providing an additional measurement on the temporal behavior of the mission, even when the system software is compromised.

Remote Attestation: Remote attestation schemes can be generally categorized into static remote attestation and runtime (or dynamic) remote attestation. Static remote attestation focuses on remotely attesting static properties of a device [14–16, 37, 66–73]. However, they are often unable to capture the dynamic behavior of the prover. Another line of work is Proof of Execution (PoX) [6–8], which can only attest whether the specified software is executed but cannot detect runtime attacks such as control flow hijacking. Recent works [3, 5, 9–12, 33, 34], including ARI, supplement these works by attesting runtime behaviors such as control flow. Lastly, while both DIAT [4] and ARI are designed for autonomous systems, their focuses are different in that ARI aims to attest RMEI, while DIAT aims to attest data integrity on messages exchanged among autonomous systems. ARI can also be combined with DIAT to further provide temporal property attestation.

Tab. 3 shows the seven closely related works. From the perspective of attestation goal, ARI differs from existing work in that it aims to attest mission integrity, which is principally derived from program behavior integrity via measurement of

Table 3: Related Work Comparison Table

System		APEX [7]	C-FLAT [3]	DIAT [4]	OAT [5]	RSWATT [67]	ScaRR [33]	ReCFA [34]	ARI
temporal property									✓
continuous attest						✓	✓		✓
PoX		✓	✓	✓	✓		✓	✓	✓
CFI			✓		✓		✓	✓	
CFI (crit. cpt. only)									✓
DFI (crit. var. only)				✓	✓				✓
Security Goal (↑)	**Optimization Techniques (↓)**								
sp	coarse info			✓					
	re-execution				✓				✓
	opt. encoding		✓					✓	✓
	part. attest						✓		
tp	selec. record								✓
	dedicate core					✓			
RT performance						✓			✓

crit. var. (critical variable), crit. cpt. (critical compartment), coarse info (coarse information), opt. encoding (optimized encoding), part. attest (partial attestation), selec. record (selective recording), sp (spatial), tp (temporal)

control flow and/or data flow events, and temporal characteristics via execution timestamps. ARI is the first attempt to measure and attest temporal properties. Furthermore, while attestation on information-flow-based program behavior is well studied [3–5], ARI is the first to attempt to combine this with continuous attestation. However, the trade-off is on the granularity of program behaviors of non-critical compartments. From the perspective of performance, existing work has adapted various optimization techniques to reduce the performance overhead. ARI is the first work to explicitly consider the real-time impact for soft real-time tasks and hard real-time tasks. ARI also does not require a dedicated processor core for continuous attestation. Lastly, to reduce performance overhead, ARI combines re-execution and optimal encoding spatial techniques from existing work [3, 5, 33, 34]. The trade-off for performance improvement is the lack of visibility in non-critical compartments.

Lastly, though ARI is related to the attack detection and prevention techniques for CPS [62, 63, 65, 74], similar to other attestation systems, ARI differs from the prevention and mitigation techniques [62, 65] in that it captures the system behaviors for remote verification.

10 Limitations and Discussions

Source Code Requirement and Certification: ARI currently assumes the availability of the source code of the targeted CPS. Even though it is possible to remove this requirement by applying binary instrumentation techniques [75], understanding the internal program structures and critical controls of a binary application may require non-trivial efforts in reverse engineering. Furthermore, similar to other security mechanisms [3–5, 31–34], the addition of security mechanisms changes the application logic as well as its performance. Depending on the agency and acceptance test procedure, the cyber-physical system may have to go through the re-certification process.

Approximation of Information Flow: The main approach in ARI to handle scalability is to trade the granularity of measurement for performance. As a result, measurements (such as CF) may not be as accurate as those in the previous works [3, 5] that capture the complete information flow. Thus, ARI can only attest over-approximation of the correct system behavior.

Manual Effort: Developing a policy that can detect mission deviation while maintaining real-time performance can be non-trivial. In general, the developer has to be familiar with the high-level software architecture of the system, and a clear assessment of the risk faced by the autonomous system during the mission. ARI embraces several design elements to improve the usability of the system. First, ARI allows for different levels of customization. Even at the lowest level of customization, ARI still attempts to minimize the real-time impact automatically. Second, ARI provides a diverse set of built-in policies for different types of systems. Our user study showed that the system is relatively usable. However, we also acknowledge that a poor choice in the policy specification will not only impact the ability to detect attacks but also the corresponding performance overhead.

Sensor (Physical) Attacks: Software remote attestation methods, such as ARI, focus entirely on the behavior in computing. They have the potential to detect system anomaly (either in timing or information flow) if sensor attacks [76] leave traces in computing, but ARI is unable to confirm the physical attacks.

11 Conclusion

In this paper, we present ARI, a real-time mission execution integrity attestation system, that continuously measures the program behaviors (both information flow and real-time properties) of the autonomous system. To tackle the prohibitive performance overhead from attestation, ARI leverages software compartmentalization to allow for meaningful measurement only on the critical compartments. The proposed system is implemented and evaluated on different policies and real-time CPS applications. The source code and extended technical report are available at our project repository[1].

Acknowledgment

We thank the reviewers for their feedback. This work is supported in part by US National Science Foundation under grants CNS-1837519, CNS-1916926, CNS-2038995, CNS-2154930, CNS-2229427, and CNS-2238635, Office of Naval Research under grant N00014-19-1-2621, Army Research Office under grant W911NF-20-1-0141, and Intel.

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

References

[1] http://cuts2.com/PZXPb.

[2] E. A. Lee, "Cyber physical systems: Design challenges," in *ISORC*, IEEE, 2008.

[3] T. Abera *et al.*, "C-FLAT: control-flow attestation for embedded systems software," in *CCS*, ACM, 2016.

[4] T. Abera *et al.*, "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems.," in *NDSS*, 2019.

[5] Z. Sun *et al.*, "OAT: Attesting operation integrity of embedded devices," in *S&P*, IEEE, 2020.

[6] I. de Oliveira Nunes *et al.*, "Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," in *ICCAD*, IEEE, 2019.

[7] I. D. O. Nunes *et al.*, "APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise," in *Security*, USENIX, 2020.

[8] A. Caulfield *et al.*, "ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems," in *DAC*, 2022.

[9] I. D. O. Nunes *et al.*, "Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution," in *DATE*, IEEE, 2021.

[10] G. Dessouky *et al.*, "Lo-fat: Low-overhead control flow attestation in hardware," in *DAC*, ACM, 2017.

[11] S. Zeitouni *et al.*, "Atrium: Runtime attestation resilient under memory attacks," in *ICCAD*, IEEE, 2017.

[12] G. Dessouky *et al.*, "Litehax: Lightweight hardware-assisted attestation of program execution," in *ICCAD*, IEEE, 2018.

[13] I. D. O. Nunes *et al.*, "Dialed: Data integrity attestation for low-end embedded devices," in *DAC*, ACM/IEEE, 2021.

[14] A. Seshadri *et al.*, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *SIGOPS Operating Systems Review*, ACM, 2005.

[15] A. Seshadri *et al.*, "Swatt: Software-based attestation for embedded devices," in *S&P*, IEEE, 2004.

[16] Y. Li *et al.*, "Viper: verifying the integrity of peripherals' firmware," in *CCS*, ACM, 2011.

[17] D. Iorga *et al.*, "Slow and steady: Measuring and tuning multicore interference," in *RTAS*, IEEE, 2020.

[18] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *RTAS*, IEEE, 2019.

[19] R. Mahfouzi *et al.*, "Butterfly attack: Adversarial manipulation of temporal properties of cyber-physical systems," in *RTSS*, IEEE, 2019.

[20] "Ardupilot." https://ardupilot.org/.

[21] G. Bernat *et al.*, "Weakly hard real-time systems," *TC*, vol. 50, no. 4, 2001.

[22] R. Wahbe *et al.*, "Efficient software-based fault isolation," in *SOSP*, ACM, 1993.

[23] L. Szekeres *et al.*, "Sok: Eternal war in memory," in *S&P*, IEEE, 2013.

[24] P. Larsen *et al.*, "Sok: Automated software diversity," in *S&P*, IEEE, 2014.

[25] M. Bechtel and H. Yun, "Memory-aware denial-of-service attacks on shared cache in multicore real-time systems," *TC*, 2021.

[26] B. Wittenmark *et al.*, "Timing problems in real-time control systems," in *ACC*, IEEE, 1995.

[27] B. Lincoln, "Jitter compensation in digital control systems," in *ACC*, IEEE, 2002.

[28] P. Marti *et al.*, "Jitter compensation for real-time control systems," in *RTSS*, IEEE, 2001.

[29] E. Ebeid *et al.*, "A survey of open-source uav flight controllers and flight simulators," *Microprocessors and Microsystems*, 2018.

[30] H. Peng and M. Payer, "USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation," in *Security*, USENIX, 2020.

[31] A. A. Clements *et al.*, "ACES: Automatic Compartments for Embedded Systems," in *Security*, USENIX, 2018.

[32] C. H. Kim *et al.*, "Securing real-time microcontroller systems through customized memory view switching.," in *NDSS*, 2018.

[33] F. Toffalini *et al.*, "ScaRR: Scalable Runtime Remote Attestation for Complex Systems," in *RAID*, 2019.

[34] Y. Zhang *et al.*, "Recfa: Resilient control-flow attestation," in *ATC*, USENIX, 2021.

[35] "Arm trustzone." http://cuts2.com/Wymiq.

[36] "Risc-v keystone." http://cuts2.com/wotsz.

[37] I. D. O. Nunes *et al.*, "VRASED: A verified hardware/-software co-design for remote attestation," in *Security*, USENIX, 2019.

[38] P. Koeberl *et al.*, "Trustlite: A security architecture for tiny embedded devices," in *EuroSys*, ACM, 2014.

[39] I. Shumailov *et al.*, "Sponge examples: Energy-latency attacks on neural networks," in *EuroS&P*, IEEE, 2021.

[40] A. Li *et al.*, "Chronos: Timing interference as a new attack vector on autonomous cyber-physical systems," in *CCS*, 2021.

[41] A. Li *et al.*, "Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference," in *RTSS*, IEEE, 2022.

[42] C. Yan *et al.*, "Sok: A minimalist approach to formalizing analog sensor security," in *S&P*, IEEE, 2020.

[43] H. Choi *et al.*, "Cyber-physical inconsistency vulnerability identification for safety checks in robotic vehicles," in *CCS*, ACM, 2020.

[44] A. A. Clements *et al.*, "Protecting bare-metal embedded systems with privilege overlays," in *S&P*, IEEE, 2017.

[45] S. McCamant and G. Morrisett, "Evaluating sfi for a cisc architecture.," in *Security*, USENIX, 2006.

[46] S. M. Silver, "Implementation and analysis of software based fault isolation," 1996.

[47] L. E. Leyva-del Foyo *et al.*, "Predictable interrupt management for real time kernels over conventional pc hardware," in *RTAS*, IEEE, 2006.

[48] "aiT." https://www.absint.com/ait/.

[49] "Carts." https://rtg.cis.upenn.edu/carts/.

[50] R. E. Kalman, "A new approach to linear filtering and prediction problems," 1960.

[51] T. Kim *et al.*, "From Control Model to Program: Investigating Robotic Aerial Vehicle Accidents with MAYDAY," in *Security*, USENIX, 2020.

[52] "Optee." https://www.op-tee.org/.

[53] "Capstone." http://www.capstone-engine.org/.

[54] D. Formby and R. Beyah, "Temporal execution behavior for host anomaly detection in programmable logic controllers," *TIFS*, 2019.

[55] P. Krishnamurthy *et al.*, "Anomaly detection in real-time multi-threaded processes using hardware performance counters," *TIFS*, 2019.

[56] "Stm32f4dsc." http://cuts2.com/GFOXl.

[57] "Lpc55s6x." http://cuts2.com/fGXCD.

[58] "Raspberry pi 3 b+." http://cuts2.com/wnqOQ.

[59] E. M. Redmiles *et al.*, "A summary of survey methodology best practices for security and privacy researchers," tech. rep., 2017.

[60] Y. Cao *et al.*, "Adversarial sensor attack on lidar-based perception in autonomous driving," in *CCS*, ACM, 2019.

[61] Y. Cao *et al.*, "Invisible for both camera and lidar: Security of multi-sensor fusion based perception in autonomous driving under physical-world attacks," in *S&P*, IEEE, 2021.

[62] R. Quinonez *et al.*, "SAVIOR: Securing autonomous vehicles with robust physical invariants," in *Security*, USENIX, 2020.

[63] H. Choi *et al.*, "Detecting attacks against robotic vehicles: A control invariant approach," in *CCS*, ACM, 2018.

[64] Y. Chen *et al.*, "Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system," in *S&P*, IEEE, 2018.

[65] A. Khan *et al.*, "M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles," in *Security*, USENIX, 2021.

[66] M. Ammar *et al.*, "SμV—the security microvisor: A formally-verified software-based security architecture for the internet of things," *TDSC*, 2019.

[67] S. Surminski *et al.*, "RealSWATT: Remote Software-based Attestation for Embedded Devices under Realtime Constraints," in *CCS*, ACM, 2021.

[68] K. Eldefrawy *et al.*, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust.," in *NDSS*, 2012.

[69] F. Brasser *et al.*, "TyTAN: tiny trust anchor for tiny devices," in *DAC*, ACM, 2015.

[70] J. Noorman *et al.*, "Sancus 2.0: A low-cost security architecture for iot devices," *TOPS*, 2017.

[71] I. De Oliveira Nunes *et al.*, "On the toctou problem in remote attestation," in *CCS*, ACM, 2021.

[72] N. Asokan *et al.*, "Seda: Scalable embedded device attestation," in *CCS*, ACM, 2015.

[73] M. Ambrosin *et al.*, "Sana: secure and scalable aggregate network attestation," in *CCS*, ACM, 2016.

[74] J. Wang *et al.*, "Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone," in *S&P*, IEEE, 2022.

[75] T. Kim *et al.*, "Revarm: A platform-agnostic arm binary rewriter for security applications," in *ACSAC*, 2017.

[76] Z. Yu, Z. Kaplan, Q. Yan, and N. Zhang, "Security and privacy in the emerging cyber-physical world: A survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 3, pp. 1879–1919, 2021.

[77] "Blake2." https://github.com/BLAKE2/BLAKE2.

[78] L. Lamport, "Proving the correctness of multiprocess programs," *TSE*, 1977.

A Case Studies on Control and SW Attacks

Control Attack: Control-level semantic attacks exploit weak input validation, making control flow recording insufficient. Analyzing control variable values is a way to detect such attacks. As delays often exist between the exploitation of a vulnerability and the manifestation of control deviation, continuous monitoring of control variable changes is necessary. Mayday [51] identifies such attack by recording the control variables and post-analyzing their runtime changes. ARI is complementary to Mayday due to (1) ARI provides a secure logging system using sandboxing, which is pre-assumed in Mayday. (2) ARI allows the logging of users' customized code blocks, providing a flexible trade-off between log granularity and runtime performance.

Attack implementation: We use one of the case studies on Arducopter in [51] as an illustrating example, where the adversary exploits a value range uncheck vulnerability by sending a MavLink message to configure the control parameter *_kp* to an abnormal value. The message is received by the function *handle_common_message()* in MavLink module, which further invokes function *handle_param_set()* to assign an abnormal value (500) to *_kp*. After that, abnormal *_kp* degrades the control performance by corrupting the velocity controllers, which finally causes the drone to crash once it makes a turn.

Detection implementation: To detect this attack, ARI adopts a policy that separates primitive controllers and input handling components. In this way, the transfers between MavLink module to the velocity controller, such as the invocation of *_kp.set_float()* in *handle_param_set()*, will be recorded. In compartments of primitive controllers, the values of critical control variables are recorded, including *_kp*. In attestation, ARI calculates the control errors by comparing reference and actual values. If there is a control digression detected, ARI can trace back the digression and find that the digression was instigated by the change of *_kp* up to 500, which confirms the root cause is the malicious message.

Software Attack - Control Flow Violation: Control flow hijacking is one of the most common software attacks. In each mission, ArduCopter will follow mission-specified waypoints. Before the mission starts, *write_cmd_to_storage()* is invoked to write mission waypoints into hardware storage. Every time a waypoint is reached, *read_cmd_from_storage()* is invoked. To conduct the case study, we manually inject a buffer overflow vulnerability in *read_block()*.

Attack implementation: We implemented the attack using a malformed waypoint in CMAC-circuit mission, a test flight mission with seven waypoints. Upon reading the malicious waypoint, the attacker hijacks the control flow of *read_block()* to deliberately crash the drone by turning off the motor using *AP_Arming::disarm()* then running an infinite loop to prevent the drone from recovering.

Detection implementation: To detect the attack, *AP_ Arming::disarm()* is assigned to a critical compartment, so malicious inter-compartment control flow from *read_block()* to *AP_Arming::disarm()* will be recorded. The run-time and memory overhead are around 4.29% and 1.91%.

B System Implementation

B.1 Compartmentalization

API and Build-in Policies: To help users to define their policies, ARI provides well-defined APIs for users to compartmentalize program and annotate criticality. Specifically, the compartmentalization APIs are designed to manipulate program on the PDG, which contains both low-level and semantic information, e.g., data-flow, source file name and function name, such that users can easily find the target functions or variables while traversing the PDG. Additionally, criticality annotation APIs allow users to label the criticality of compartments and variables. For more details, please refer to the developer guide in the open-sourced repository.

B.2 Program Instrumentation

Compartment Isolation: ARI restricts both control and data flow either by masking the address with reserved registers or by using trampolines. Key instrumentation instructions are shown in Tab. 4. Specifically, both intra-data/control flows are restricted by address masking with the reserved register r_{rsv}. To reduce the overhead of restricting the data flows, red zones instead of address masking are used to restrict local write instructions that use *sp* relative addressing. For inter-compartment data/control flows, trampolines are used to allow cross-stack data access and inter-compartment transfers, as well as record the control transfer targets. Fixed mask modification is used to access shared variables. Since a control flow can be either inter or intra-compartment flow, distinguishing them can be more time-consuming than masking.

Therefore, stub wrappers are added at function returns for inter-compartment callers to reduce the time of distinguishing return addresses.

Table 4: Instrumentation in ARI for Cortex-A platform

Type	Original Instruction	Sanxbox Instruction
intra-forward indirect jump	bx/blx rx	bfi r_{rsv}, rx, #0, #n bx/blx r_{rsv}
intra-backward indirect jump	pop pc, bx lr	pop lr (only for pop pc) bfi r_{rsv}, lr, #0, #n bx r_{rsv}
inter-compartment transfer	bx/blx rx	bfi r_{rsv}, rx, #0, #n cmp rx, r_{rsv} push lr (only for bx) blne tpl_switch_cpt bx/blx r_{rsv}

	b/bl #addr	movt r_{rsv}, #addr
		movw r_{rsv}, #addr
		push lr (only for b)
		bl tpl_switch_cpt

	pop pc, bx lr	pop lr (only for pop pc) bfi r_{rsv}, lr, #0, #n bx r_{rsv}
intra-forward direct function call	bl func	bl func_internal
data access	str/ldr ry, [rx]	bfi r_{rsv}, rx, #0, #n str/ldr ry, [r_{rsv}]

Real-time Attested Event Measurement: Real-time mission measurement mechanism implementations in ARI include compiled time instrumentation insertion for recording and runtime storage and sealing during mission execution.

Recording Function Insertion: CFEventsPASS is used to insert trampolines to record control flow events according to the encoding mechanism in Appendix B.3. To record the temporal property of critical compartments, *TPEventsPASS* identifies entries and exits of the critical compartment whose temporal property needs to be measured according to policy to insert timestamp recording. To efficiently calculate the hash of return addresses, ARI uses BLAKE2 [77] hash algorithm.

Runtime Storage and Sealing: During mission execution, events are measured according to the encoding scheme illustrated in Appendix B.3, and stored in memory which is outside the sandbox. Meanwhile, a sealer signs the measurements and stores them in storage periodically. To prevent compromised applications from corrupting the measurement sealing process, ARI thus runs the sealer in TEE. To reduce runtime overhead introduced by frequent world switches, ARI leverages batching and multi-threading mechanisms to reduce the number of world switches. Since the sealer is implemented as a real-time task, during a mission, the execution time is measured as the worst-case execution time of sealing and then storing a batch of measurement results on hardware storage, and the period is measured as the shortest period of filling a batch of the measurement buffers.

Table 5: Inter-Compartment Event Encoding Scheme

Inter-cpt Transfer	Direct Jmp	Indirect Jmp	Reture
Critical->Critical	none	dst_addr	dst_addr
Critical->Noncritical	none	dst_addr	dst_addr
Noncritical->Critical	dst_loc_id	dst_addr	dst_addr
Noncritical->Noncritical	dst_cpt_id	dst_cpt_id	des_cpt_id

Table 6: Intra-Critical-Compartment Encoding Scheme

Direct Jmp	Indirect Jmp	Return	Cond. Branch
none	des_addr	hash	true/false

B.3 Recording Overhead Minimization

Measurement Event Encoding: ARI encodes control flow information as shown in Tab. 5, taking a few approaches to reduce data overhead. First, ARI only records the destination of any CF event since the source is already recorded as the preceding CF event's destination. Second, ARI doesn't record the destination of a direct jump sourced from a critical compartment because it can be inferred during intra-compartment CF attestation. Third, when possible, ARI encodes compartment and location ID that takes fewer bytes instead of full address information. For example, for noncritical to critical CF events, each critical compartment only has limited and fixed addresses transferable from other compartments, allowing ARI to record the destination location ID instead of the address. Similarly, ARI doesn't attest CF events inside noncritical compartments, only recording their compartment IDs. As shown in Tab. 6, ARI attests all control flow inside critical compartments, recording their destination addresses of the indirect jumps, the hash of the return addresses, and conditional branch decisions.

Recording with SFI and Lock-free Ring: Existing techniques store the measurement metadata in memory either by trapping them into a TEE [3,5] or in privilege mode [31]. Context saving and restoring during system trapping is expensive, as is shown in Fig. 1. To minimize the performance penalty, ARI makes use of the isolation provided by the software sandboxing to store metadata securely while avoiding context switching overhead caused by system trapping. Specifically, ARI saves mission measurement data outside the sandboxes into a lock-free ring buffer [78]. Meanwhile, a mission events sealer runs as a separate thread that signs and stores mission events into storage. To minimize CPU utilization, the mission events sealer fetches and signs mission events in batches. Introducing new tasks, i.e. measurement sealer may break real-time schedulability of real-time CPS. To maintain real-time CPS schedulability, ARI implements the mission measurement sealer as another real-time task in CPS, and integrates it into schedulability analysis.

Table 7: Real-time Tasks Execution time

App	Task	Runtime w/o ARI	Runtime w ARI	Deadline
SP	Injec. (A)	109 ms	119 ms	2000 ms
	Injec. (M)	114 ms	132 ms	2000 ms
OC	PSA	4.536 ms	4.722 ms	100 ms
	SR	131 us	139 us	25 ms
	DO	952 us	986 us	100 ms

SP(Syringe Pump), OC(Oxygen Concentrator), Injec.(Medicine Injection), PSA(Pressure Swing Adsorption), SR(Sensor Read), DO(Device Operation)

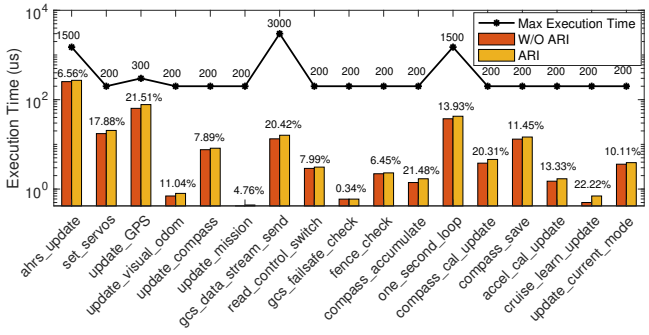


Figure 8: Tasks Execution Runtime Overhead in ArduRover (controller-based policy with fail-safe controller as critical)

C Evaluation

Tasks Deadline Miss Rate: Tab. 7 and Fig. 8 shows that real-time tasks in ArduRover (AR), syringe pump (SP), and oxygen concentrator (OC) do not exceed their deadlines even under the policy with the highest runtime overhead from Section 7.1. The real-time performance of house alarms was not measured as it is not a real-time application.

Timing Recording Overhead: To measure runtime overhead under different timing recording granularity, we measure tasks/operations average runtime overhead of all five applications on both Cortex-A and Cortex-M under three kinds of configurations, including recording timestamps on every entering and exiting critical compartment, on every control-flow transfer between different compartments, and every control flow event in the whole system. The average tasks/operations execution time is measured by recording the timestamps at the beginning and end of each task/operation and calculating the difference. As shown in Fig. 9. Timing recording in ARI generates the highest overhead on ArduCopter among the five applications because of its complexity. Specifically, recording all control flow events and all compartment transfer generates 470% and 7.4% runtime overhead respectively. Recording compartment transfer to and from critical compartment only generates 2.27% runtime overhead.

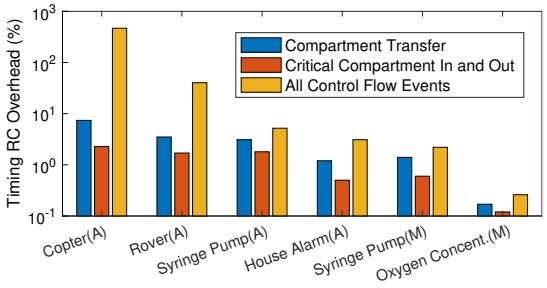


Figure 9: Time Recording Runtime Overhead

D Additional Security Analysis

CF/DF Attestation Security: To evade ARI, an attacker can *1. Disable program instrumentation:* However, code integrity is guaranteed by memory protection mechanisms as described in Section 3. Any instrumentation bypassing by control flow hijacking is recorded and will be detected in the verification phase. *2. Manipulate program Behavior measurement functions:* However, the program behavior measurement trampolines are located separately from every compartment. Intra-compartment indirect jumps can not jump to program behavior measurement trampolines because of the sandbox. Inter-compartment indirect jumps to program behavior measurement trampolines will be recorded and verified in the verification phase. *3. Hijack control flow by generating a return address hash collision:* Hijacking control flow to generate a hash collision is at least as hard as finding a hash collision. This can very challenging when a collision-resistant hash algorithm is used.

Log Manipulation: An attacker may try to manipulate the control flow or critical data read-write recording mechanism to manipulate what ARI records. There are three ways to manipulate ARI recording systems maliciously. Attackers can *1. Manipulate the logging function or TEE API to sign and seal mission information onto the disk directly:* ARI prevents such attack by putting the logging trampoline which has access to the log ring buffer and TEE API outside sandboxes. Any invocation to the trampoline uses direct jump instruction. Thus, an attacker in the sandbox cannot jump to the middle of the trampoline function by using intra-compartment indirect jump instructions. Exploitation by inter-compartment indirect jumps is recorded and verified. *2. Modify signed log record stored on disk to change the content of the log:* This is prevented by ARI using AES to encrypt logs. *3. Modify the mission events stored in plaintext in DRAM temporarily:* However, ring buffers can only be accessed by trampolines which are also outside sandboxes.