

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ARTO: Efficient Execution Integrity Attestation for Real-Time Operation of Cyber-Physical Systems

Ruizhe Zhao, Cong Sun, and Zongzhen Li, *State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University;* Tiantian Wang, *Northeastern University;* Yunbo Wang, *State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University*

https://www.usenix.org/conference/usenixsecurity26/presentation/zhao-ruizhe

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

ARTO: Efficient Execution Integrity Attestation for Real-Time Operation of Cyber-Physical Systems

Ruizhe Zhao[1], Cong Sun[1]*, Zongzhen Li[1], Tiantian Wang[2], and Yunbo Wang[1]
[1]*State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University*
[2]*Northeastern University*
[1]*{ruizhezhao@stu.xidian.edu.cn, suncong@xidian.edu.cn},* [2]*wang.tianti@northeastern.edu*

Abstract

Real-time embedded systems are prevalent in cyber-physical applications such as drones and autonomous vehicles. Due to their high complexity, the environmental uncertainties, and the openness to remote control, these systems are vulnerable to control-flow hijacking and data-only attacks that compromise runtime integrity and reliability. Traditional control-flow integrity (CFI) and data-flow integrity (DFI) approaches impose excessive overhead, while the control-flow attestation (CFA) allows detection latency incompatible with real-time constraints. This paper presents ARTO, the first control-flow and data-flow protection approach that integrates the control-flow attestation to provide strong security with competitive runtime overhead. Through the prover-side path segmentation and hash-based validation result caching, ARTO provides partial context-sensitive protection at minimal runtime cost, offloading complete context-sensitive detection to the remote verifier. Furthermore, ARTO implements an equivalence-class-based data-flow protection that combines the address-based target check for memory reads with the value-based checks to protect critical variables, stronger than the SOTA operation integrity enforcement in validating the read source for critical variables. Extensive evaluations on robotic vehicle autopilots confirm ARTO's efficacy in mitigating both control-flow and data-only attacks while maintaining runtime overhead within real-time limits, outperforming SOTA CFA and operation-integrity enforcement methods in operation execution time.

1 Introduction

Real-time embedded systems typically feature complex control-loop logic and interact with the external environment via sensors and actuators. Many studies, e.g., [17,24,30], have revealed the vulnerabilities of such systems to control-flow hijacking and data-flow tampering, posing serious threats to their runtime behaviors.

*Corresponding author.

To mitigate control-flow hijacking, various *control-flow integrity* (CFI) approaches [6,28,34,36,37,39,40,47,53] have been proposed. These approaches typically target general-purpose programs and ensure system security by validating the conformance of runtime execution traces against a predefined control-flow graph. Despite modern CFI approaches having achieved significant advancements in precision and performance [18], there are still challenges in applying them to real-time embedded systems, e.g., the costly runtime refinement of protection [36,53] and the underlying hardware features [28, 34, 37, 47] that are unavailable on embedded platforms. Dedicated efforts have been proposed to enforce CFI for real-time or embedded systems [10,35,46,48,51,56]. These approaches often trade off between performance and security. The lightweight approaches [10, 35, 56] typically prioritize protecting backward edges. In contrast, the systems that enforce comprehensive context-sensitive protection [46,48,51] incur prohibitive overheads, e.g., up to 6x [48], which are unaffordable by the maximum execution time limit of systems like autopilots.

An obvious limitation of the local CFI defenses is that they fail to provide evidence for remote supervision on the mission's execution integrity. Such remote supervision is typically necessary for systems like robotic vehicle autopilots, while the local CFI leaves the system states opaque during their autonomous operations. *Control-flow attestation* (CFA) [7] establishes a root of trust for the remote verifier to rigorously monitor the prover device's system state and forces the prover to generate cryptographic evidence of its actual execution flow, thus effectively providing reliable control-flow diagnoses. The online overhead of CFA methods is primarily tied up by the measurement on the prover, making several of these approaches [7,45,49] applicable to embedded systems. However, CFA methods are allowed to report the control-flow violation asynchronously in post hoc. Generally, deploying both local control-flow integrity enforcement and the control-flow measurement for remote diagnosis at the same time is impractical due to the unacceptable overhead on embedded devices. The only work that addresses both requirements is

CFA+ [11], relying on the ARM Branch Target Identification (BTI) hardware feature, which is unsupported on the autopilot boards of robotic vehicles. However, for systems like autopilot, local and immediate detection of control-flow hijacking remains valuable, as such a compromise could lead to targeted attacks on infrastructure or the public, far more serious than merely crashing the drone. Therefore, an efficient mechanism that can both locally detect the control-flow hijacking and provide control-flow evidence to a remote trusted party is essential.

Data-only attacks can manipulate critical decision-making data without violating the static control-flow policy, thereby evading detection by both CFI and CFA. Such attacks have proven practical in embedded systems [16, 22, 45]. The *data-flow integrity* (DFI) mechanism is supposed to enforce predefined data dependencies and mitigate such attacks [20]. Full DFI for all memory reads and writes incurs 2x-3x performance overhead and is impractical for embedded systems. Alternatively, recent efforts [31, 45, 50] have focused on protecting a portion of memory reads and writes for efficiency. In these approaches, the value-based integrity checks [31, 45] provide more versatile protection than the address-based check that relies on memory tag extension (MTE) and BTI hardware features [50]. However, these value-based monitoring methods still fail to verify the target addresses of memory reads, leaving the adversary the opportunity to manipulate address registers and exploit them.

Motivated by the above requirements, we propose ARTO, the first method that enforces CFI and CFA while simultaneously protecting the data flow of critical variables for the critical operations in real-time systems. To balance security guarantees and performance overhead, ARTO segments the control-flow graph into multiple sub-paths. ARTO ensures partial context-sensitive CFI at the end of each sub-path by validating the sub-path against the pre-measured hash database. The complete context-sensitive verification is offloaded to the remote verifier by ARTO's CFA mechanism, which reconstructs the complete execution history using the reported sub-path hashes. To mitigate data-only attacks, ARTO leverages static alias analysis to group potentially aliased critical variables into equivalence classes. The runtime data-flow verification first validates the memory target using the equivalence-class identifier, then performs value-based detection to ensure data-flow integrity. To meet the real-time constraint, ARTO makes extraordinary efforts to reduce the runtime overhead. First, to accelerate the local verification process, ARTO caches the validated sub-path hashes in a protected memory region. Second, ARTO adopts the *software fault isolation* (SFI) mechanism to secure the local evidence by confining the write operations to a designated region, rather than protecting the evidence in the TEE as OAT [45] does. To confine the memory writes, ARTO uses basic-block-level analysis to aggregate and simplify the instrumented security checks compared to naive SFI implementations. We make the following contributions:

- ARTO is the first integrity protection framework that integrates CFI with CFA and is tailored for the real-time operations of cyber-physical systems. ARTO achieves efficient local verification by offloading the context-sensitivity of control-flow protection to CFA and caching local validation results, outperforming the SOTA CFA approaches for embedded systems [45, 49] by reducing the execution time by 17.6% to 94.2%.
- ARTO's equivalence-class-based data-flow checking mechanism combines the address-based target check for memory reads with the value-based check, providing more rigorous protection for critical variables than the SOTA value-based checking [45]. ARTO outperforms other data-flow protection [50] by reducing runtime overhead by 35.3% to 86.1%.
- We demonstrate the practicality of ARTO via extensive evaluation on the real-world CPS (ArduPilot and PX4), verifying its effectiveness to defend against both control-flow and data-only attacks while ensuring the runtime overhead remains within the control loop cycle limits.

2 Background

2.1 Control-Flow and Data-Flow Protections in Real-Time Systems

Control-flow integrity techniques [6] validate execution traces with a control-flow graph (CFG) by inline monitoring and restricting control transfers to valid targets. Several comprehensive context-sensitive protections have been developed on embedded systems [46, 48, 51]. RECFISH [48] achieves fine-grained forward-edge protection but relies on specific hardware features (e.g., Cortex-R virtualization) that are absent in typical cyber-physical systems. Its frequent privilege switching under recursion can cause up to 6x overhead. Sherloc [46] uses the macro trace buffer (MTB) to collect non-sequential control-flow transfers, and runs the trace configuration, storage, and validation inside the TEE. Its holistic enforcement can also delay non-secure interrupts during trace validation, making it less suitable for time-sensitive tasks. To further improve precision, InsectACIDE [51] implements context-sensitive CFI via hardware tracing. However, high-frequency branch execution generates excessive trace data, resulting in a significant slowdown. The MTB feature used by [46, 51] is also unsupported by the autopilot hardware.

Although many works [7, 8, 14, 26, 27, 41, 45, 49, 54, 55] have focused on measuring control-flow events on the device and diagnosing control-flow policy violations remotely and asynchronously, such post-hoc diagnosis is deficient to real-time embedded systems. For example, in a regular flight, the drone autopilot conducts the attitude control loops at 400 Hz. If an adversary hijacks the control flow to trigger a `motor_disarm` command, the drone would immediately lose stability and crash. In contrast, the experimental results of SOTA CFA [49] for real-time systems show that the verifier's verification la-

```
1  // Simplified control loop for robotic arms
2  void main_looper() {
3      int st = 0;
4      cmd_t cmd;                    // Command from remote controller
5      int (*op_func)(int, char*);   // Pointer to operation
6      while(1) {
7          if (st = read_command(&cmd)) {
8              get_input();                                  //BUGGY!
9              if (st) {     //Reads cmd.op under data-only attack
10                 op_func = get_op_func(cmd->op);
11                 (*op_func)(cmd->p_size, cmd->param);
12             }
13         } ...
14     } ...
15 }
16 void get_input(){
17     char  peripheral_input[PERIPHERAL_INPUT_SIZE];
18     ...
19     gets(peripheral_input);
20     ...
21     return ;
22 }
23 // The corresponding assembly code:
24 00000714 <get_input>:
25  714:   push  {fp, lr}                        //Store caller's fp
26  718:   mov fp, sp
27  71c:   sub sp, sp, #8
28  720:   sub r0, fp, #3
29  724:   bl  4e8 <gets@plt>    // Data-only attack: Buffer overflow
        modifies on-stack fp to (fp-8)
30  728:   mov sp, fp
31  72c:   pop {fp, pc}                  //Load (fp-8) to caller's fp
32 000009cc <main_looper>:
33  ...
34  a08:   sub r0, fp, #36               //cmd locates at fp-36
35  a0c:   bl  730 <read_command>
36  a10:   str r0, [fp, #-24]                    //st's def site
37  a14:   cmp r0, #0
38  a18:   beq a6c <main_looper+0xa8>
39  a1c:   bl  714 <get_input>
40  a20:   ldr r0, [fp, #-24]                    //st's use site
41  a24:   cmp r0, #0
42  a28:   ble a58 <main_looper+0x94>
43  a2c:   ldr r0, [fp, #-32]               //cmd.op's use site
44  a30:   bl  6d0 <get_op_func>
45  ...
```

Listing 1: Example of Robotic Arm's Control Loop Vulnerable to Data-Only Attack

tency may exceed 10 minutes for the ArduCopter autopilot. Another more persistent attack example is to intentionally flip the return target of `updateOutputs` and hijack the control flow of PX4's `mixer_module` to reduce the output frequency of a certain actuator and the Pulse-Width Modulation (PWM) level of a specific rotor on the drone. The reduced PWM level of the rotor could lead to unpredictable system states or even targeted attacks on the public until the remote verifier identifies this attack in minutes. In such cases, the local CFI remains valuable, as the immediate detection of the control-flow hijacking could trigger a timely failsafe for the drone.

Moreover, the non-control-data attacks [21] are also practical and impactful on embedded systems. The adversary corrupts the critical non-control data, such as configuration values, identity attributes, or key decision variables, and leverages these modified values to manipulate program behavior. Such manipulations introduce no illegal control-flow transfer, and the CFI and CFA protections fail to detect them. The traditional data-integrity protections [9, 19, 20] instrument *all* memory operations, rendering them impractical for the resource-limited embedded systems. Consequently, recent protections have moved toward selectively safeguarding only security-critical variables [31,45,50], with value-based checking [31,45] most suitable for autopilot hardware. By verifying value consistency between definition and use sites rather than tracking the validity of every memory access address, this method significantly reduces overhead while meeting real-time constraints and ensuring schedulability. The value-based checking approaches [45] cannot ensure that a variable's use actually reads from the intended memory location; thus, they fall short of identifying data-only attacks that manipulate the read locations of critical variables. An example is presented in Listing 1. In this example, both `st` and `cmd` are critical variables. The buffer overflow in `get_input` modifies the `fp` register to (`fp`−8). When the `main_looper` intends to read `st` (Line 9), because the target address of the read instruction intended for `st` has been redirected to the field `op` of `cmd`, the statement at Line 9 actually reads the value of `cmd.op`. The value-based check receives the address of `cmd.op` along with its valid value. Since this address-value pair is legitimate, the check passes. However, the conditional branching at Line 9 is erroneously determined by the value of `cmd.op` rather than `st`, thereby compromising the execution integrity. The most straightforward solution to this issue is to combine address-based checks with value-based checks. Before verifying value consistency, the protection should determine whether the memory read instruction accesses the correct memory location based on a pre-generated data-flow graph (DFG). However, this rigorous address tracking incurs extremely high overhead and is unsuitable for embedded systems.

Combining the local CFI and DFI for real-time embedded systems compels us to give more consideration to the trade-off between performance and security. The first observation is that applying the fine-grained protection over the entire system is impractical. Our local integrity enforcement should be conducted on the safety-critical functional components of the system. In another dimension, we make more trade-offs between the strength of protection techniques and runtime overhead. Specifically, for the CFI enforcement, we retain only limited context sensitivity at the device and offload the complete context-sensitive control-flow violation detection to a post-hoc control-flow attestation. For data-flow protection, we enhance the value-based checking with a prior address-based target checking for memory reads. This enhancement is more efficient than the DFG-based rigorous address tracking. By statically organizing the variables with the same alias into an equivalence class and capturing the valid def-addresses of these variables at runtime, ARTO can detect the compromised read location (address of `cmd.op` in Line 40 of Listing 1) not in the set of valid def-addresses of `st`'s equivalence class, thus detecting the data-only attack. As a result, we summarize the brief comparisons between the SOTA related works and ARTO, as presented in Table 1.

Table 1: Comparison between SOTA Approaches and ARTO (●: Strong; ◐: Medium; ○: Weak)

Approach	CFI	CFA	DF Protection	RT-support	Isolation
RECFISH [48]	◐	✗	✗	●	MPU-based
SHERLOC [46]	◐	✗	✗	◐	TEE-based
InsectACIDE [51]	●	✗	✗	●	TEE-based
ARI [49]	✗	◐[1]	○	●	SFI-based
OAT [45]	✗	◐[1]	◐	✗	TEE-based
OP-DFI [50]	✗	✗	◐	●	MTE-based
ARTO	◐	●	●	●	SFI-based[2]

[1] Not support exact attacking position diagnose at verifier side.
[2] Require a TEE-protected secure timer.

2.2 Threat Model

Following prior works [7, 45, 49, 54], we assume that the platform enforces *data execution prevention* (DEP), preventing adversaries from injecting code into the target program space or tampering with the instrumented code. Previous studies [23, 38] have demonstrated that such code-integrity enforcement is practical even in embedded systems. Additionally, we assume the presence of a secure timer to support the periodic randomized self-check mechanism proposed in Section 3.3.

We consider a powerful adversary with complete knowledge of the memory layout and arbitrary memory read and write primitives targeting within the memory. This capability allows the adversary to corrupt function pointers or return addresses residing in writable memory regions to hijack the control flow of either the application code or ARTO's instrumented code. Furthermore, the adversary can manipulate critical data variables to execute data-only attacks without altering the control-flow graph. We exclude denial-of-service (DoS) attacks, physical tampering, and side-channel attacks from our scope, as these threats are orthogonal to the control-flow hijacking and data manipulation considered in this study.

3 System Design

For the flexibility to trade off security and performance, ARTO's protection can be enforced across the entire embedded system or only for specific *critical operations* of the system. Typically, the boundary of critical operations is determined using domain-specific knowledge. The control-flow graph inside the boundary of critical operations could have one or more connected components. Each connected component is defined as a critical operation. The overall workflow of ARTO consists of an offline phase and an online phase, as depicted in Fig. 1 and Fig. 2, respectively.

The offline phase of ARTO integrates multiple analyses and instrumentation to build the binary and measurement databases for online use (Fig. 1). ARTO first builds a precise inter-procedural control-flow graph for each critical operation (abbreviated as OpCFG) on the LLVM-IR of the target program. In this process, the indirect calls are properly resolved. Guided by the OpCFG, ARTO performs static analyses and instrumentation at both the LLVM-IR level and the assembly level to enforce 1) the local CFI and protected prover-side measurement of CFA (**C** in Fig. 1), 2) the local data-flow protection (**D** in Fig. 1), and 3) the software-fault-isolation (SFI) based protection (**S** in Fig. 1). More specifically on the three features:

- For the CFI and CFA (Section 3.1), ARTO identifies the *verify points* that separate the control-flow paths into *path segments* and serve as instrumentation points for the *verify engine*. The path segmentation ensures partial context sensitivity within a path segment (defined in Appendix A.3), while offloading the complete context-sensitive control-flow protection to the CFA verifier. The instrumented verify engine code conducts the local CFI verification. The *control-flow events* used in the CFI and CFA are mainly the indirect transfers (indirect calls and returns) within the target program. ARTO's IR-level analysis also selects appropriate conditional branches as control-flow events to help the remote verifier recover the complete execution trace for the context-sensitive CFA. The control-flow event recordings are instrumented at the assembly level for efficiency, guided by an IR-assembly event-mapping mechanism (Section 4). Both CFI and CFA require offline measurements. With the instrumented binary and the OpCFG, ARTO's binary-level *simulated executor* explores all possible execution paths and obtains control-flow measurements, which are embedded in the reserved privileged space of the instrumented binary for local CFI and stored by the remote verifier as the *hash-trace map* for CFA.
- For the data-flow protection (Section 3.2), ARTO uses field-sensitive and context-insensitive data-flow analysis to establish the points-to relations for variables in the OpCFG. From the initial set of *critical variables* (i.e., those in branch conditions or manually tagged as critical), ARTO expands the critical variable set with the points-to relations and data dependencies. ARTO then obtains the *May-aliases* relations to group the critical variables into *equivalence classes*. ARTO properly instruments at the definition sites (def-sites) and use-sites of each critical variable with *definition table* updating or checking operations to enforce runtime integrity.
- ARTO's SFI-based protection (Section 3.3) enforces strict boundaries to prevent unauthorized access to privileged code and data regions. Specifically, ARTO determines appropriate memory writes to instrument the SFI check that avoid writing to protected memory space. Beyond this data-flow isolation, ARTO also inserts `bfi` instructions before the indirect transfers to isolate the target program's control flows from the privileged code.

In the online phase (Fig. 2), the prover-side execution enforces the local CFI and data-flow protection while preparing the control-flow evidence for the remote CFA. Upon reaching a verify point, the verify engine validates the hash digest of the

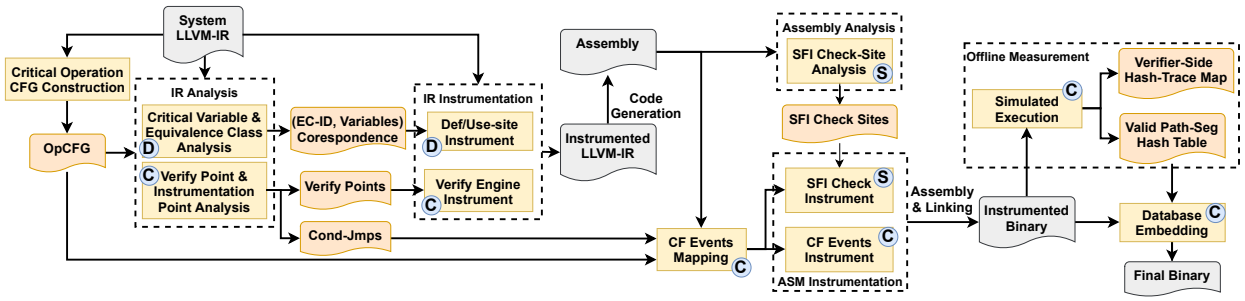


Figure 1: ARTO Offline Phase

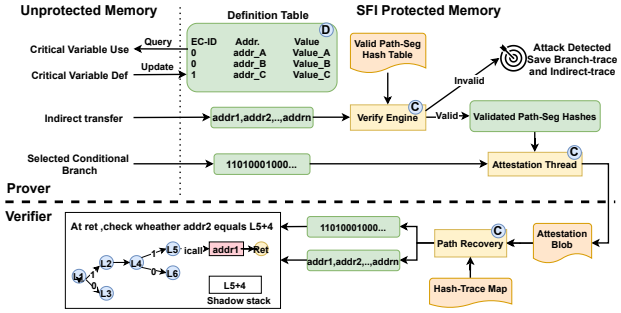


Figure 2: ARTO online Phase

current path segment against the embedded hash table. A validation failure triggers an immediate alert and the transmission of control-flow evidence to the verifier. Conversely, the engine stores and caches valid hash digests for the path segments. The *attestation thread* periodically transmits these stored digests, along with the recorded conditional branching bits, to the verifier. Regarding data-flow protection, at each def-site, the prover captures the address, value, and equivalence class ID of the critical variable by updating the corresponding entry in the definition table. At the use-site, the prover validates the *read target address* based on its assigned equivalence class ID and ensures that the value stored in the definition table entry matches the actual value read from memory. On the other hand, the verifier uses the received hash digests and conditional jump information to reconstruct the complete execution trace. It then simulates execution on the binary along this path to verify complete context-sensitive control-flow integrity.

3.1 Critical Operation Control-Flow Integrity

A naive integration of the CFI check and CFA measurement at every indirect control-flow transfer will exhaust the resources of real-time embedded systems, and is thus both unsustainable and unnecessary. To improve performance, ARTO relaxes the local CFI enforcement to be only partially context-sensitive. We separate the control-flow paths of the OpCFG into path segments. The determination of path segments relies on the selection of the verify points in the OpCFG. Specifically, a path segment, denoted $ps(v_1, v_2)$, consists of all control-flow paths from verify point v_1 to v_2, without any other verify point in between. The path segment is directed thus $ps(v_1, v_2)$ and $ps(v_2, v_1)$ are independent of each other. The local verify engine at the verify point v_2 enforces strict intra-path-segment control-flow integrity for $ps(v_1, v_2)$ by cumulating a sequence of runtime-reached indirect-branch targets into a hash digest and validating it against a pre-measured hash database. This design preserves the context within the indirect-target sequence without requiring a local shadow stack. At the same time, the reconstruction of the complete execution trace is offloaded to the remote verifier for detecting sophisticated backward-edge attacks.

This section first elaborates on the selection of the verify points for determining path segments (Section 3.1.1), then presents the details of local measurement and verification (Section 3.1.2) and the remote verification (Section 3.1.3).

3.1.1 Verify Point Selection

The location of the verify points is critical to ARTO's efficiency. The objectives of the automated selection procedure are twofold: 1) preventing path explosion caused by cyclic control flows, and 2) trading off between local verification granularity and runtime overhead.

Handling Cyclic Control Flows Loops and recursion introduce cyclic paths that in theory can generate infinite execution traces. To separate these paths into finite sub-paths in the path segments, ARTO decides the minimal set of the verify points for breaking the cyclic structures as follows:

- Loop: We designate the entry of each loop's latch [1] basic block as a verify point. If the loop body is executed, the latch block denotes the completion of an iteration. Therefore, this choice ensures that each loop iteration is verified, effectively preventing path explosion between verifications.
- Recursion: Unlike loops, recursive cycles involve both forward and backward transfers. Consequently, the verify engine should be inserted at both the entry and exit points of selected recursive tail functions. As illustrated in Fig. 3, designating `FuncD` or `FuncB` as recursive tail functions would incur redundant checks. To eliminate this redundancy, we propose the strategy (Algorithm 1 in Appendix A) to calculate the *cycle appearance count* of each node and iteratively select the highest-weighted node to break the maximum number of cycles. This process identifies the optimal in-

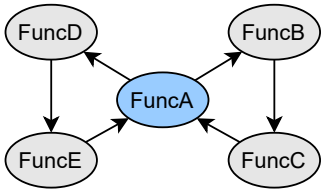


Figure 3: Recursive Call Graph Example

strumentation target, i.e., `FuncA` in this case. Note that Algorithm 1 only breaks the edges `FuncA`→`FuncB` and `FuncA`→`FuncD` via inserting verify points at the entry and exit points of `FuncA` without disrupting the graph connectivity.

Optimizing Verification Frequency via Clustering After breaking the recursion cycles to obtain the *call tree* of the critical operations, ARTO further chooses appropriate *leaf functions* of this tree and inserts verify points at their entries. These verify points help reduce the length of the control-flow event sequence between verify points, effectively balancing the size of path segments and the local verification frequency. Instrumenting all leaf functions is unnecessary, since the leaf functions called sequentially by a single caller could result in too short path segments and too frequent verifications. Therefore, ARTO uses a clustering-based algorithm to identify tightly coupled leaf functions within the call tree. Specifically, leaf functions are first sorted in a list L in descending order by their distance to the entry of the critical operation. Then, the clustering procedure iteratively operates on L. For the first function f in L (farthest to the critical operation entry), ARTO finds the leaf functions that have an edge-distance $\leq \Delta$ from f (specifically, $\Delta = 3$ in our implementation). These leaf functions are grouped into the cluster of f, and this cluster's functions are removed from L. This procedure iterates until L is empty. In each cluster of leaf functions, ARTO selects only the leaf function farthest from the critical operation entry to insert the verify point.

3.1.2 Local Measurement and Verification

Recovering the complete context information of the prover's execution trace at the CFA verifier requires a selective recording of the prover's conditional branches. This section first presents the conditional-branch selection approach for ARTO's instrumentation and defines the offline measurements. Then we elaborate on the runtime control-flow event measurement and the local verification by the verify engine.

Conditional Branch Selection ARTO measures two categories of control-flow events: *conditional branches* and *indirect control-flow transfers* (specifically *indirect calls* and *returns*). To handle indirect jumps, which are largely introduced by the `switch` statements via jump tables, ARTO's compiler uses the jump-table disabling optimization (`-fno-jump-tables`) to lower the indirect jumps into explicit conditional branches in the LLVM-IR.

To minimize runtime overhead, only a subset of the conditional branches will be selected as control-flow events for ARTO's measurement. ARTO employs a recursive reachability analysis to filter out conditional branches. Specifically, for a conditional branch, ARTO examines the subsequent control-flow paths stemming from both the *true* and *false* conditions, and traces them to the nearest function exit point. If the reachable blocks along these paths contain neither function invocations nor function pointer assignments, the branch is excluded. Crucially, this check is recursive: if a subsequent block contains another conditional branch, the analysis continues to traverse its paths. This procedure ensures that only the conditional branches that can impact the execution of attestation-relevant inter-procedural operations are instrumented.

Offline Measurement Definition ARTO's partial context-sensitive CFI relies on an offline-generated *valid path segment hash table*. Meanwhile, a *hash-trace map* is required for remote CFA. We develop a binary-level *simulated executor* that explores all valid paths of a critical operation starting from the operation entry. The implementation details are presented in Appendix A.2. With the executor, we obtain the valid path segment hash table $\mathcal{D} : \mathcal{V} \times \mathcal{V} \mapsto \mathcal{H}^*$ and the hash-trace map $\mathcal{M} : \mathcal{H} \mapsto \mathcal{T}^*$, where $\mathcal{V}$ is the verify point address space, $\mathcal{H}$ is the hash value space, and $\mathcal{T}$ is the target address space of indirect control-flow transfers. Each item $(v_i, v_j, \{hash_1, \ldots, hash_k\})$ in $\mathcal{D}$ holds all the valid accumulated hashes of the path segment $ps(v_i, v_j)$. Each item $(hash_i, (tarAddr_1, \ldots, tarAddr_m))$ in $\mathcal{M}$ holds the correspondence between the accumulated hash of a control-flow sub-path and the sequence of indirect transfer target addresses encountered on this sub-path.

Measurement Scheme At runtime, ARTO records execution traces into two separate buffers based on the instruction type. For the instrumented conditional branches, ARTO records the execution outcome as a 1-bit value (0 or 1) indicating whether the condition is met, and stores it into the *trace buffer* (TBuf). Meanwhile, for each indirect control-flow transfer, ARTO first captures the raw target address and appends it to the *indirect buffer* (IBuf). When the execution reaches a verify point, ARTO processes the contents of the IBuf to generate a cumulative digest. Specifically, the measurement procedure at the verify point iterates over the recorded target addresses using a sequential hash update, i.e., $H = \text{Hash}(H \oplus tarAddr_i)$, where $\oplus$ denotes the concatenation operation. This mechanism produces a hash digest representing the indirect control-flow history for the current sub-path within the path segment. The IBuf may be empty at a specific verify point; in this case, ARTO's verify engine will skip the hash accumulation and the subsequent local verification at this point.

Local Verification with Caching Optimization After obtaining the accumulated hash digest for the current sub-path at the verify point, the verify engine matches this hash digest against the embedded database of $\mathcal{D}$. The database (Table 8)

```
v0    int main() {                                  // Operation Start
        callfunc(1, 2);
tar1    /* irrelevant code */
        callfunc(a, b);
tar2    /* irrelevant code */
v1      return 0;                                   // Operation End
      }
      void callfunc(int a, int b) {
        int c = func(a, b);
tar3    /* irrelevant code */
ret1    return c;
      }
v2    int func(int a, int b) {                      // Leaf Function
       a += b;
       /* irrelevant code */
ret2    return a;
      }
```

Listing 2: Example of Local Verification and Remote CFA

is indexed by the path segment, under which all valid sub-path hashes of the path segment are stored. The verify engine uses the current path segment under execution and the hash digest to query the database. A successful match indicates a legitimate sub-path is being executed. ARTO resets the IBuf and deposits the hash digest into another *hash buffer* (called HBuf), allowing a low-priority attestation thread to asynchronously transmit the evidence to the remote verifier. HBuf can support CFA to reconstruct execution trace to provide context sensitivity between sub-paths, which will be discussed later in Section 3.1.3. Otherwise, the verify engine detects a control-flow hijacking attempt, and either triggers immediate program termination or activates the failsafe mechanism (for autopilots). Meanwhile, ARTO immediately transmits the contents of IBuf, TBuf, and HBuf to the verifier, thereby facilitating post-hoc analysis of the root cause.

To reduce the local verification overhead, ARTO introduces a cache-based optimization. ARTO maintains another *hash cache* for the verified sub-paths. This cache resides in the SFI-protected region and enables constant-time hash lookups. The first-time query of a valid hash digest in $\mathcal{D}$ will cause this digest to be cached. Each runtime hash digest should reach a query in $\mathcal{D}$ only if it got missed in the hash cache.

3.1.3 Remote Verification

Although the local verification based on the sub-path hashing is highly efficient, it is inherently incapable of enforcing inter-sub-path context sensitivity. This limitation allows an advanced adversary to manipulate the control flow between sub-paths without being detected by the verify engine, as individual hash digests remain valid. Considering the example in Listing 2, v_0, v_1, and v_2 are verify points. The indirect edges are (ret_1, tar_1), (ret_1, tar_2), and (ret_2, tar_3). According to the offline measurement generation strategy (Appendix A.2), the offline hash database of this program should be

$$\mathcal{D} = \{(v_0, v_2, \emptyset); (v_2, v_2, \{h_1\}); (v_2, v_1, \{h_2\})\}$$

such that $h_1 = \text{Hash}(\text{Hash}(tar_3) \oplus tar_1)$ and $h_2 = \text{Hash}(\text{Hash}(tar_3) \oplus tar_2)$. On the other hand, the hash-trace map at the verifier side is

$$\mathcal{M} = \{(h_1, (tar_3, tar_1)); (h_2, (tar_3, tar_2))\}$$

When the program runs as intended, the sub-paths along the execution trace encounter the following control-flow events:

- Sub-path from v_0 to the first reach of v_2 encounters no control-flow event.
- Sub-path from the first to the second reach of v_2 encounters (ret_2, tar_3) and (ret_1, tar_1).
- Sub-path from the second reach of v_2 to v_1 encounters (ret_2, tar_3) and (ret_1, tar_2).

The verify engine at the second reach of v_2 can match the runtime cumulative digest to h_1 of $(v_2, v_2, \{h_1\})$ in $\mathcal{D}$, while at v_1, the cumulative digest can be matched with h_2 of $(v_2, v_1, \{h_2\})$ in $\mathcal{D}$.

An adversary can hijack the return edge (ret_1, tar_2) after the second invocation of `callfunc()`, redirecting this transfer to (ret_1, tar_1). This hijacking causes a third reach of v_2, where the newest cumulative digest still matches with h_1 of $(v_2, v_2, \{h_1\})$ in $\mathcal{D}$. Since this attack does not generate any invalid sub-path hashes, the local verification is successfully evaded. To address this limitation, ARTO leverages control-flow attestation (CFA) to reconstruct the complete execution trace and perform global context-sensitive re-execution. At runtime, each sub-path hash h_i along the execution trace is stored in the HBuf. When the attestation thread is scheduled, it transmits the sequence HBuf $= (h_1, h_2, \ldots, h_n)$ along with a nonce, signed using the device's attestation key K_{att}:

$$\left(\text{Sign}_{K_{\text{att}}}(\text{HBuf}, \text{nonce}), \text{HBuf}, \text{nonce}\right)$$

to the verifier. The verifier first validates the signature to ensure authenticity. It then reconstructs the complete execution trace via the pre-established hash-trace map. For the example in Listing 2, the regular execution transmits HBuf $= (h_1, h_2)$ to the verifier, and the verifier recovers the *indirect-target trace* $(tar_3, tar_1, tar_3, tar_2)$ with the hash-trace map $\mathcal{M}$. In contrast, the hijacked execution transmits HBuf $= (h_1, h_1, \ldots)$ to the verifier, and the verifier recovers $(tar_3, tar_1, tar_3, tar_1, \ldots)$ with $\mathcal{M}$.

The purpose of the verifier-side re-execution is to restore inter-sub-path context dependencies and to verify the indirect control-flow events. The verifier performs this simulated re-execution on the target binary. The verifier maintains a global shadow stack during the re-execution. Guided by the reconstructed indirect-target trace and the received binary sequence of TBuf, it simulates the prover-side execution. At the entry of the critical operation, the shadow stack is empty. When a selected conditional-branch instruction is encountered, the verifier consumes a 1-bit record from the binary sequence of TBuf. The value 1 directs the simulation to the *true*-branch, while 0 directs the simulation to the *false*-branch. Not that

at the conditional branch instruction unselected as a control-flow event, either branch could be taken since there will be no difference in the attestation. Whenever the re-execution encounters a function call, the verifier pushes the corresponding return address onto the shadow stack, and the control flow is directed to the callee. If the function call is an indirect call, the verifier consumes the headmost target of the recovered indirect-target trace to find the callee. Upon encountering a return instruction, the verifier also consumes an indirect target from the recovered trace and checks whether this target matches the top element of the shadow stack. For the Listing 2 example under the above control-flow hijacking, when the second invocation of `callfunc()` returns, the top element of the shadow stack is tar_2, whereas the corresponding indirect target in the reconstructed trace is tar_1. This discrepancy demonstrates that the return edge has been tampered with and the prover-side execution is invalid.

3.2 Data-Flow Integrity

To enforce DFI for critical operations, we propose a def–use integrity model that combines value-based check with address-based check using memory-region equivalence classes.

Offline Static Analysis The ARTO compiler first automatically identifies critical variables annotated by developers or in branch conditions to form the initial critical variable set. Ensuring the integrity of these variables requires a global analysis of all the variables and pointers that can affect these variables through data dependencies or points-to relations. To identify the complete critical variable set, OAT [45] has proposed a fixpoint algorithm but did not release its implementation in the repository [5]. ARTO independently implements the fixpoint algorithm (depicted in Appendix A.4). After obtaining the complete critical variable set, ARTO refines the global points-to relations (obtained via the Andersen pointer analysis) to the points-to information related to the critical variables. ARTO further performs a May-aliases analysis based on the points-to relations to partition the critical variables into equivalence classes. Specifically, if variables *A* and *B* are respectively an alias of variable *C*, then variables *A*, *B*, and *C* are grouped into the same equivalence class. Once the equivalence classes are determined, the ARTO compiler assigns a globally unique identifier (*SetID*) to each equivalence class. The compiler instruments different operations for all the critical variables. At each def-site of a critical variable, ARTO instruments the recording operation before storing the value to the critical variable. At each use-site of a critical variable, ARTO instruments an equivalence-class-based read target check, combined with the value-based check, after loading from the critical variable. The instrumented code takes the corresponding *SetID* as a parameter.

Runtime Enforcement To enforce data-flow protection, ARTO maintains a two-dimensional array within the protected memory region. We refer to this structure as the *definition table* (DT), which organizes data based on the equivalence class and memory address. Specifically, the table works as a direct-mapped cache. The row index is the static equivalence class identifier (*SetID*). The column index is derived from the variable's physical address (*Addr*) using a modulo mask (i.e., $Addr \,\%\, \text{MASK}$). Each cell in the table, denoted as $\text{DT}[SetID][Addr \,\%\, \text{MASK}]$, stores the most recently validated value of the critical variable whose memory address is *Addr* and equivalence class ID is *SetID*.

The runtime enforcement triggers distinct operations based on the variable's lifecycle. At def-sites of writing some value *Val* to some critical variable at *Addr*, the instrumented monitor captures the tuple $(SetID, Addr, Val)$, calculates the indices from *SetID* and *Addr*, and directly updates the corresponding cell in DT with *Val*. At use-sites of a critical variable with *SetID*, if the program is reading *CurrVal* from address *CurrAddr*, the instrumented monitor captures the tuple $(SetID, CurrAddr, CurrVal)$ after the variable is loaded, then retrieves the def-site value from the table using the indices (*SetID* and $CurrAddr\%\text{MASK}$), and compares the def-site value with *CurrVal*. An incorrect *CurrAddr* will retrieve an incorrect table cell in the row of *SetID*, whose def-site value is inconsistent with the read value *CurrVal*, causing an integrity violation. Recalling Listing 1, when the adversary causes Line 40 to read from `cmd.op`, because `cmd.op` has a different *CurrAddr* from `st`, ARTO will retrieve another table cell in the row of `st`'s *SetID* for the value comparison, leading to the integrity violation. The only thing a data-only attacker can do is to modify the read target to another critical variable that holds the same value. Such a compromise cannot cause any control-flow circumvention.

Besides, if the critical variable is global and defined beyond the boundary of the critical operation, then at the boundary of the critical operation, ARTO instruments configurable value-range checking for the def-site operation of the critical variable. In our experiments, we use dynamic analysis on the autopilots in software-in-the-loop (SITL) simulations to profile the proper value ranges for these boundary def-sites, while in general, these value-range decisions are beyond our scope of data-flow protection.

3.3 Lightweight Software-Based Isolation

ARTO's prover designates the following components as *privileged*: 1) measurement and check functions, 2) attestation thread, 3) local verify engine, 4) runtime evidence (including definition table), and 5) the offline-generated hash database. The first three components are *privileged code*, while the last two are *privileged data*. Unlike existing schemes that rely on TEE-based isolation (Table 1), ARTO avoids frequent domain-switching overhead by embedding these components into the target program's address space. ARTO's SFI-based protection uses a customized linker script to allocate a dedicated, runtime-protected region within the program's address space

to hold all privileged components. To ensure the integrity of privileged components against attacks originating from non-privileged code, ARTO SFI adopts the following security objectives: ⓐ Preventing adversaries from invoking privileged code via indirect control-flow transfers. Privileged code can be reached only via statically authorized direct calls. ⓑ Preventing unprivileged code from modifying privileged data, including runtime evidence, the hash database, and other protected structures. © Preventing adversaries from bypassing the verify engine and thereby evading control-flow checks.

Control-Flow Integrity of Privileged Code To achieve the objective ⓐ, ARTO restricts the targets of all the indirect control-flow transfers, avoiding any branch into the protected runtime region. ARTO performs control-flow validation at designated verify points rather than at every indirect branch. While this design improves efficiency, it introduces a brief temporal window during which an adversary might attempt to redirect execution into privileged code. Therefore, global restrictions on the address range are mandatory.

The ARTO compiler places the target program's code and data segments in the lower address space, whereas the protected runtime region is isolated at higher addresses. At each indirect transfer, ARTO instruments a runtime sanitization step that masks the high bits of the transfer target using the bit-field insert (`bfi`) instruction, effectively confining the transfer target to the unprivileged address space. Specifically, a reserved register r_{rsv} stores the final target of the indirect control-flow transfer. The register r_{rsv} is initialized to zero and used as `bfi`'s target, while the original target register serves as the source. Table 9 in Appendix A.5 summarizes the representative instruction-level transformations used in this protection.

Data Isolation To achieve the objective ⓑ, ARTO ensures that privileged data can only be modified by privileged code. Therefore, ARTO restricts the target address of all memory write instructions. Conventional SFI mechanisms typically insert an address-validation check at each store instruction, resulting in substantial runtime overhead. To improve efficiency, ARTO introduces a novel SFI optimization that eliminates redundant address checks for consecutive memory writes with a single basic block. The key insight is that, within a single basic block, if a register serves as the base address for multiple store instructions and is not modified throughout the instruction sequence, the base register remains invariant. Therefore, it is sufficient to instrument a single address-range check after the final store instruction in the sequence. If the last write is verified secure, all preceding writes that share the same unmodified base register are secure as well. Placing the address-range check after the last store prevents attackers from hijacking control flow to bypass the check. We develop a per-basic-block analysis to identify the memory writes that need to be instrumented (as described in Appendix A.5). The inserted checking logic is presented in Table 9. At runtime, the inserted invocation of the store-address-check function takes the base register as input and verifies whether this write-target address lies outside the protected region. If a write targets a privileged area, ARTO raises a security exception.

Moreover, to prevent adversaries from exploiting large immediate offsets (`#imm`) for out-of-bounds writes, ARTO places redzones at both the start and end of the protected memory region. Each redzone's size equals the maximum possible offset value, ensuring that even extreme offset manipulations cannot reach privileged data.

Periodic Randomized Self-Check To enforce ©, ARTO introduces a *periodic randomized self-checking* (PRSC) mechanism to ensure that the instrumentation code, particularly the instrumented calls to the verify engine, has not been bypassed at runtime.

Complex real-time embedded systems, such as autopilots, are multithreaded. The critical operations are typically conducted as (a portion of) a specific thread (task). For the critical operations in this scenario, we define two protected functions, *start_measure* and *exit_verify*, to indicate the beginning and end of a critical operation. They are respectively inserted at the entry and exit points of the target critical operation. At runtime, *start_measure* can set a thread-local flag to indicate that the critical operation is in progress and notify the privileged code to act as for the critical operation. The function *exit_verify*, which is a special verify engine function, will unset this flag. The thread-local flag can distinguish an execution of a critical operation from the uncritical use of its functions.

Another observation is that the critical operations in this scenario are typically required to be executed at least once in each control loop (e.g., control loop cycle $\delta = 2.5$ ms for ArduCopter). PRSC extends the *start_measure* and *exit_verify* functions and uses a secure-timer-triggered checking to periodically confirm that the verify-engine bypassing can be detected. When an attacker tries to bypass the call to the verify engine at a verify point, the bypass will be captured at the subsequent verify points in the execution trace and detected by the verifier. The situation becomes complicated when the attack circumvents *all* subsequent verify engine calls in the execution trace starting from a verify point. In this case, the abnormal control-flow events are cumulated in IBuf and TBuf but not reflected in HBuf, thus remaining unknown to the verifier. Since this attack circumvents the call to *exit_verify*, we extend *exit_verify* to update a protected random value $rand_x$. We use the secure timer to trigger a TEE-protected periodic check for the changes to $rand_x$ every δ interval, ensuring *exit_verify* is not bypassed in this loop. TEE holds the last value of $rand_x$. Upon the timer's cycle expiration, the processor is forcibly trapped into the check procedure to read the latest $rand_x$ from the privileged region and verify its freshness. If this periodic check detects bypassing of *exit_verify*, it will send the contents of IBuf and TBuf to the verifier. The hardware timer guarantees the strict checking frequency. Note that only privileged code can configure the secure timer via a specific interface in TEE's secure world, rendering the timer inaccessible to unprivileged code. An-

other situation is that the *start_measure* function is bypassed, so the verify engines are not notified to work for the critical operation. To resolve this issue, we also extend *start_measure* to update another protected random value $rand_e$, and perform a similar secure-timer-triggered check to alert the verifier.

4 Implementation

We built the prototype of ARTO on the ARMv7-A architecture Raspberry Pi 3B board, LLVM 16.0 for analyses and instrumentations, and Capstone [43] for disassembly by the offline simulated executor and verifier. ARTO's hash operation uses BLAKE2s [4] by default and also supports xxHash [3] via compile-time macros for efficiency, with performance differences presented in Section 6.1. To build the OpCFG, we perform program analysis to construct the inter-procedural control flow graph (ICFG) of the target program. Building a precise ICFG requires resolving indirect call targets. We combine multiple static analyses. We use LLVM's type metadata [2] for virtual function resolution, analogous to the class inheritance analysis. For indirect calls of function pointers, we use the intersection of the results from SVF [44] and DEEPTYPE [52] to obtain a more accurate and convergent set of indirect targets. After constructing the ICFG and specifying the entry of the critical operation, we perform a depth-first traversal to generate the OpCFG.

We extend LLVM-16 for ARTO's analyses and instrumentations related to control flow, data flow, and SFI. For control-flow protection, we implement two LLVM frontend passes and a backend pass. The first frontend pass identifies verify points and inserts verify engine calls. The second fronted pass identifies the conditional jumps that need to be recorded and inserts dummy code at indirect call sites and selected conditional jumps. The dummy code facilitates the control-flow event mapping in Fig. 1. Specifically, since LLVM metadata is discarded during code generation and absent at the assembly level, we insert an IR instruction `call icall_dummy_<id>` at the indirect call site, where <*id*> is the unique ID of the indirect call we assigned as metadata. For selected conditional jumps, the pass inserts the IR instruction `call cond_call_dummy` before the conditional jumps. These dummy calls cannot be removed during code generation, but will be removed by the following backend pass. The backend pass inserts trampoline functions to record control-flow events based on the presence of dummy code. Then, the backend pass removes the dummy calls of conditional jumps. For the dummy calls of indirect calls, the backend pass retrieves the specific indirect call ID from the inserted `call icall_dummy_<id>`, and replaces this dummy call with a special assembly instruction, `movw` r_{rsv}, <*id*>, after the indirect call site. These `mov` instructions are used in different ways by the offline simulated executor and the verifier. When the executor reaches an indirect call, it extracts <*id*> from the `mov` instruction to look up the valid target set within the OpCFG. The verifier uses these indirect call IDs to validate the IBuf content sent by the secure-timer-triggered check.

5 Security Analysis

Under our threat model (Section 2.2), to evade ARTO, an attacker would need to: ❶ bypass the execution of the verify engine to escape inspection, ❷ modify the runtime evidence stored in the protected region, ❸ generate a hash collision via control flow manipulation to deceive the local verification mechanism, or ❹ hijack control flow into the privileged code to induce malicious behavior from privileged modules.

Regarding ❶, an attacker attempts to hijack control flow to skip the verify points and the corresponding verify engine calls. ARTO's PRSC mechanism uses a hardware-secured timer to trigger periodic checks, ensuring that bypassing verify points (especially the ending *exit_verify*) can be detected within a control loop and that compromised control-flow evidence can be received and detected by the verifier.

To defend against ❷, we isolate all privileged data (including the offline hash database and runtime evidence) in a dedicated runtime-protected region. ARTO's optimized SFI mechanism ensures that the contents of this region cannot be modified by any write instruction within the target program.

Regarding ❸, an attacker attempts to manipulate the control flow to generate a hash collision with the hash of a valid sub-path to deceive the local verification mechanism. However, hijacking control flow to generate a hash collision is at least as difficult as finding a collision in the underlying hash function. ARTO's collision-secure hash algorithm [4] makes such collision construction computationally infeasible. ARTO's performance-prioritized hash algorithm [3] primarily focuses on accidental or naive control-flow manipulations that are more pervasive and can also be feasible to crash the drone.

For ❹, an attacker may attempt to use an indirect transfer to execute privileged code. ARTO's strict control-flow isolation ensures that privileged code resides in the high address space, and the `bfi` instruction is inserted before each indirect transfer instruction. This instruction uses a reserved register r_{rsv} to mask the high bits of the target address, strictly confining all indirect transfer targets to the unprivileged region. Consequently, privileged code can be accessed only via direct calls, and attackers cannot take malicious action within the privileged code by hijacking indirect branches.

6 Evaluation

We evaluate ARTO on a Raspberry Pi 3 Model B platform, featuring a Cortex-A architecture with a 1.2 GHz CPU and 1GB of RAM. The ARTO compiler and the verifier are deployed on a workstation with a 2.50 GHz×12 Intel(R) Core(TM) i5-12400 CPU, 16GB RAM, and Linux 5.15.0-139-generic kernel (Ubuntu 20.04). To comprehensively evaluate the per-

Table 2: Target Programs and Critical Operations

Target	#Func	Description
ArduCopter [13]	179	`update_xy_controller` is the main critical operation entry. This operation serves as the core navigation logic translating horizontal position errors into desired roll and pitch angles. Its integrity is essential, as any compromise in this logic would generate erroneous velocity commands, leading to uncontrollable drifting or catastrophic crashes.
ArduRover [13]	30	`AC_P_2D::update_all` is the critical operation entry. This operation serves as the core of the position controller, calculating the corrective velocity vector required to minimize position error. Its integrity is critical, as tampering with this logic would generate erroneous guidance commands, causing vehicle to deviate from intended trajectory.
PX4 [42]	69	`PositionControl::_velocityControl` is the critical operation entry. This operation calculates velocity updates based on the time interval `dt`. It is essential for maintaining the drone's dynamic stability. Any manipulation of this velocity loop disrupts flight maneuvers, leading to immediate loss of control or crashes.
SP HA LC RM	12 5 2 107	Embedded apps from OAT [45], i.e., Syringe Pump (SP), House Alarm (HA), Light Controller (LC), and Remote Movement Controller (RM). Strictly following critical operation selections and configurations of OAT for comparison.

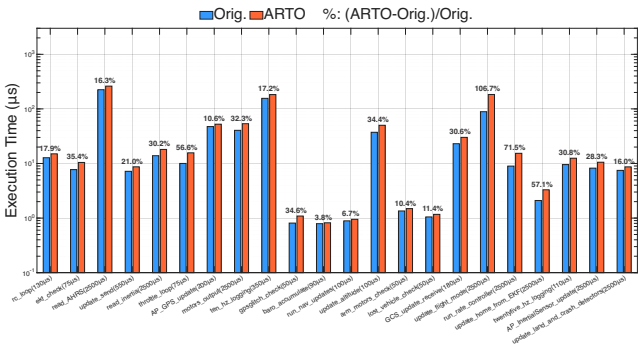
formance of ARTO, we use seven target programs, i.e., three real-world complex CPS systems and four lightweight embedded apps. The detailed target programs and the number of functions in the selected critical operations are presented in Table 2. For the mission-critical flight and ground control systems, our critical operation choices focus on the high-frequency control loops that are most sensitive to runtime overhead. To ensure robust protection, in addition to the primary control variables, we annotate local variables within these critical operations as critical variables to enforce comprehensive data-flow integrity.

Our evaluations answer the following research questions:

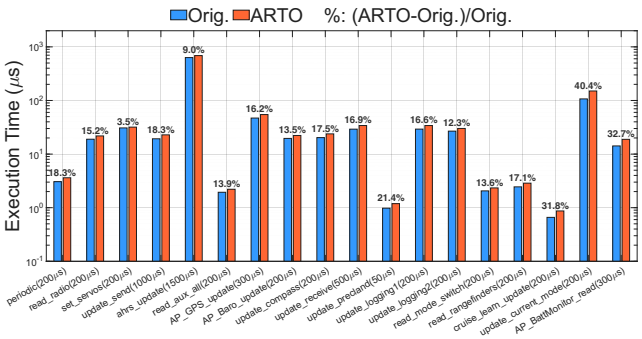
- RQ1. What are the execution time and memory footprint overhead of ARTO?
- RQ2. Can ARTO effectively mitigate control-flow hijacking and detect data-only attacks?
- RQ3. What are the boundaries of the protection provided by ARTO in complex cyber-physical systems?
- RQ4. What is ARTO's real-time impact on the autopilot?

6.1 Runtime Overhead (RQ1)

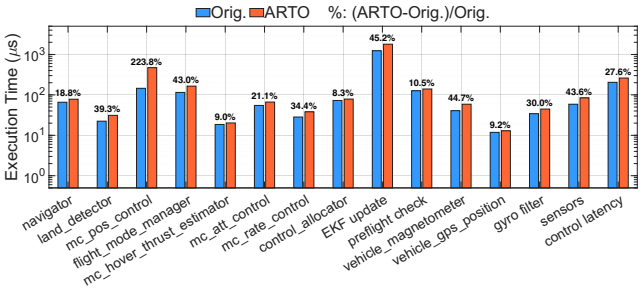
Each system runs representative workloads for evaluation. Specifically, ArduCopter and ArduRover run over the standard CMAC-circuit trajectory flight mission for one hour, while PX4 executes the `vtol_mission` flight mission for one hour. To measure ARTO's execution-time overhead on autopilots, for ArduCopter and ArduRover, we record the duration of each main control loop running at high frequency. For PX4, we use the motor output cycle to represent the control loop cycle. We set the secure timer cycle of ARTO to the maximum main control loop cycle (typically 2.5 ms for ArduCopter and ArduRover and 4 ms for PX4). We calculate the average duration of the control loop cycle with and without ARTO's protection. For the lightweight embedded apps, we run each



(a) ArduCopter (Expected Task Execution Time Marked in Task Name)



(b) ArduRover (Expected Task Execution Time Marked in Task Name)



(c) PX4 (Max Loop Cycle Typically 4 ms)

Figure 4: Tasks Execution Time Overhead

app 100 times and measure the average execution time of the attested operations.

Execution Time Overhead As detailed in Table 3, ARTO introduces an average runtime overhead of 24.3% when using BLAKE2s. Specifically, for the autopilots, the average overhead is 40.1% to the execution time of each loop. To ensure ARTO-protected autopilots meet the expected task-wise execution time requirements, we monitor the per-cycle execution time of each autopilot's functional tasks. The average per-cycle execution times are presented in Fig. 4. We can clearly observe that under ARTO's protection with BLAKE2s, all tasks maintain average per-cycle execution times much lower than their expected execution times (PX4 has only a monolithic reference limit, i.e., 4 ms). Specifically, for ArduCopter, the task `update_flight_mode` exhibits the highest execution overhead, due to our explicit protection of the critical operation `update_xy_controller` within it. For

Table 3: ARTO Runtime Overhead

Target Program	Prover-Side Execution Time					Prover-Side Mem Size(KB)		Instrumentation and Analysis Statistics								Evendence Size (KB)	Verifier-Side Verification Time (s)
		xxHash		BLAKE2s				#Branch Cond		#EC	#Def-Use		#Indirect	SFI #Write			
	Orig.(μs)	+ARTO(μs)	Overhead	+ARTO(μs)	Overhead	Orig.	+ARTO	Orig.	CF-Evnt		Def	Use	Transfer	Orig.	Instru.		
ArduCopter	522	741	42.0%	815	56.1%	6,780	14,461	211	149	93	313	502	128	9,627	5,897	537,395.21	1,092.6
ArduRover	1,106	1,239	12.0%	1,276	15.4%	6,372	14,892	32	28	19	20	52	24	8,924	5,486	20,228.50	129.6
PX4	340	428	25.9%	506	48.8%	5,344	13,772	68	52	63	236	319	60	17,683	9,913	306,997.30	844.8
SP	173,305	174,821	0.9%	174,997	1.0%	560	696	9	9	16	31	52	8	17	17	63.43	7.2
HA	1,394	1,659	19.0%	1,672	19.9%	580	692	73	54	36	77	210	7	28	20	0.17	0.1
LC	5,379	6,841	27.2%	6,927	28.8%	572	712	0	0	12	23	72	0	7	2	0.89	1.0
RM	10,019,911	10,022,248	0.02%	10,026,700	0.1%	600	688	510	385	134	643	1,616	71	418	383	7.68	3.4

ArduRover and PX4, the tasks `update_current_mode` and `mc_pos_control` respectively incur the most significant overhead for the same reason. In contrast, other tasks incur overhead primarily from ARTO's SFI-based protection.

Memory Footprint Overhead Table 3 reveals a correlation between app complexity and memory footprint overhead. For the complex autopilot firmware, the memory footprint increases by around 110% to 160%. These substantial increases are primarily due to the size of the hash database and our empirical configuration of the buffer sizes (IBuf, TBuf, and HBuf) for capturing control-flow events. In contrast, the smaller apps incur much lower memory overheads (<25%).

Instrumentation Statistics Table 3 also demonstrates the effectiveness of our static analysis in reducing instrumentation overhead. Regarding control-flow protection, our analysis effectively prunes unnecessary conditional branches from being monitored as control-flow events. For instance, in ArduCopter, only 149 of 211 conditional branches are required to be instrumented. This reduction is achieved by filtering out conditional jumps that do not impact the control flow. In complex CPS logic, the subsequent logic of specific conditional branches mainly performs mathematical computations without impacting the control flow; such branches are deemed structurally irrelevant and are thus excluded from control-flow events.

The equivalence classes metric (#EC) highlights the efficiency of ARTO's DFI. To optimize instrumentation, we merge the critical variables into equivalence classes by assigning identical identifiers to the definition and use sites within each class. This approach effectively consolidates the massive data dependencies of ArduCopter into 93 classes, thereby significantly simplifying the runtime checking of def-use chains.

To clarify the efficacy of basic-block-level analysis of ARTO's SFI protection, the CPS firmware frequently employs complex data structures for chunk operations, leading to sequences of `str` instructions that utilize invariant base registers within a single basic block for indirect addressing. By simplifying the store-address checks, we significantly reduced the instrumented code and overhead. Notably for PX4, this optimization eliminated approximately 44% of redundant store checkpoints (from 17,683 candidates to 9,913 essential sites).

Evidence Size and Verification Time The transmission overhead is decided by the size of the control-flow evidence. ARTO records hash values for indirect control-flow transfers at sub-path granularity and uses a conditional-branch trace to guide the verifier's re-execution, leading to a relatively large volume of control-flow evidence logs. As shown in Table 3, ArduCopter produces more than 500 MB of evidence log in a one-hour flight mission. However, this overhead yields significant forensic benefits. Compared to OAT [45] and ARI [49], ARTO's logging strategy enables the verifier to pinpoint the exact location of control-flow hijacking attacks precisely (Table 1). Regarding verification time, ARTO can reach a peak verification throughput of around 491 KB/s. Both the evidence size and verification time fit the normal network speed and avoid storage explosion at both sides.

Table 4: Overhead Comparison with OAT and ARI

Program	OAT Time(μs)	OAT/ARTO	ARI Time(μs)	ARI/ARTO
ArduCopter	N/A	N/A	1,450	1.78x
ArduRover	22,126	17.34x	1,548	1.21x
PX4	N/A	N/A	N/A	N/A
SP	875,434	5.00x	177,289	1.01x
HA	354,524	212.04x	19,102	11.42x
LC	832,859	120.23x	156,132	22.54x
RM	11,332,204	1.13x	10,024,329	1.00x

Comparison with SOTA CFAs We compare BLAKE2s-based ARTO against OAT and ARI. To ensure fair comparison, we modify ARI's partitioning strategy to perform software compartmentalization at the critical operation level, while OAT inherently supports operation-based attestation. As Table 4 illustrates, OAT incurs the highest overhead. This is primarily because OAT requires frequent TEE world switches to process control-flow events. Specifically, the excessive latency introduced by these switches causes timing violations, rendering OAT's deployment on ArduCopter infeasible. Moreover, we observed that OAT and ARI are unable to support PX4 due to C++ standard compatibility issues. ARTO's outperformance over ARI is attributed to two factors: lightweight SFI protection and filtering redundant conditional jumps. In contrast, ARI incurs higher overhead by recording all inter-compartment control-flow events. This intensive logging rapidly exhausts the evidence buffer in complex systems like ArduCopter and ArduRover, leading to the inevitable loss of control-flow evidence. On autopilots, ARTO achieves 17.6%-94.2% execution time reduction than OAT and ARI. Comparisons with BLAST [54] are shown in Appendix B.

Comparison with Other DFI We compared the overhead of ARTO's data-flow protection with OP-DFI [50]. To ensure a fair comparison, we excluded ARTO's CFI protection by turning off the related instrumentation. We attempted to deploy OP-DFI on the autopilots in Table 2. However, OP-DFI did not release the symbolic analysis entry file for these au-

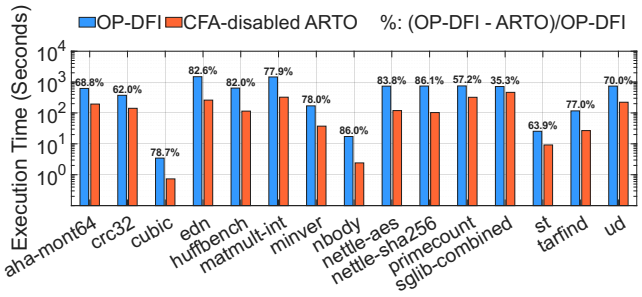


Figure 5: Overhead Comparison with OP-DFI

topilots, and its data-flow analysis cannot terminate on the autopilot tasks that locate the critical operations in Table 2. Consequently, we use the Embench suite [29] for this evaluation. We instrument all variables in the target programs. This setup highlights the efficiency of our equivalence class and value-based checking mechanisms. Fig. 5 details the comparison results. Overall, ARTO improves OP-DFI in runtime overhead by 35.3% to 86.1%.

6.2 Attack Mitigation Performance (RQ2)

This section evaluates the efficacy of ARTO in safeguarding critical operations against different attacks.

Attack Development We use both manual vulnerability injection (as done by ARI [49]) and public CVEs (i.e., CVE-2025-15150 and CVE-2025-5640 on PX4) to evaluate the effectiveness of attack detection.

To construct the control-flow hijacking, we inject a stack buffer overflow into `read_uint16()` in ArduCopter. By uploading a crafted *malformed waypoint* containing a payload that exceeds the local buffer capacity, an adversary can corrupt the stack frame and return address, thereby launching an attack during mission execution. In this way, we construct three attack scenarios: ① overwriting the return address within the stack frame to divert control to an arbitrary offset inside the critical operation, following an undefined path in CFG; ② crafting a payload that redirects execution to the entry of `AP_Arming::disarm()`, aiming to immediately cut motor power; and ③ manipulating the control flow to return to an alternative call site of `read_uint16()` within the critical operation, which constitutes a valid path in CFG. These attacks are injected into the vulnerable critical operation `read_cmd_from_storage` of ArduCopter. Besides, we develop attack scenario ④ within the program `aha-mont64` to demonstrate entire-program protection. We also develop attack scenario ⑤ on PX4 to exploit the stack buffer overflow in `handle_log_request_data()` reported in CVE-2025-15150, and hijack its return target. We craft a malicious log file with an excessively long filename and use MAVLink log request to trigger this control-flow hijacking in the vulnerable critical operation `handle_message`.

We develop three data-only attack scenarios. Attack scenario ⑥ injects the Robotic Arm's control loop program (Listing 1) with the attack described in Section 2.1 to modify the on-stack frame pointer (`fp`) and cause the read operation for `st` to erroneously retrieve the value of `cmd.op`. Attack scenario ⑦ exploits stack overflows to modify critical variables respectively in `aha-mont64`, ArduCopter, and the Listing 1 program. Specifically, in the Listing 1 program, we exploit the stack overflow vulnerability in `get_input` to modify only the critical variable `st`'s value. We set its value to 0, causing the `if`-condition to evaluate to false and the program to terminate. In attack scenario ⑧, we exploit the stack buffer overflow of CVE-2025-5640. We craft an MAVLink message to trigger an out-of-bound write to the local buffer `trajectory_waypoint` in the vulnerable critical operation `handle_message_trajectory_representation_waypoints`. This out-of-bound write is controlled by the critical variable `number_valid_points` identified by ARTO.

Table 5: Comparison of Control-flow Hijacking Detection

Target Program (Attack ID)	Method	Local		Verifier	
		Detected?	Time	Detected?	Time
ArduCopter (Attacks ①-③)	ARI	No	N/A	No	N/A
	OAT	No	N/A	No	N/A
	ARTO (attack ①)	Yes	$< 1\,\mu s$	Yes	1.07 s
	ARTO (attack ②)	Yes	3.14 ms	Yes	3.89 s
	ARTO (attack ③)	No	N/A	Yes	9.62 s
`aha-mont64` (Attack ④)	ARI	No	N/A	Yes	76.66 s
	OAT	No	N/A	No	N/A
	ARTO	Yes	$< 1\,\mu s$	Yes	0.91 s

Effectiveness against Control Flow Hijacking ARTO mitigates the attacks ①-⑤ through distinct mechanisms. Regarding attacks ①, ④, and ⑤, as execution proceeds, the program inevitably encounters a verify point. Since the control flow follows an undefined path, ARTO's local CFI detects the anomaly. For attack ②, because the control-flow redirection circumvents the verify points at the exit points of `read_cmd_from_storage` and `update_xy_controller`, the protected random value stops updating, triggering ARTO's PRSC mechanism to issue a timeout alert, which takes longer than detecting attack ①, as shown in Table 5. ARTO's verifier detects attack ③. Although the exploited return path exists in the CFG, this redirection violates the global context-sensitivity constraints enforced by ARTO.

Table 5 also compares ARTO's detection effectiveness with that of other CFAs [45, 49]. Despite attacks ① and ④ being of the same category, ARI can only detect attack ④ at the verifier side. ARI fails to detect the attacks ①-③ in ArduCopter because its runtime evidence buffer is rapidly exhausted due to the high frequency of control-flow events. While OAT is designed to detect these attacks, the verification engine in the released implementation [5] omits certain hash computations, thereby preventing successful detection. In these experiments, we calculate the time from attack initiation to the local violation detection to emphasize the importance of local CFI. On the other hand, when measuring the detection time cost of CFA approaches, we include the time from the attack initiation to the completion of evidence production, as well as

Table 6: Comparison of Data-Only Attack Detection

Target Program (Attack ID)	Method	Local	
		Detected?	Time
aha-mont64 (Attack ⑦)	OP-DFI	No	N/A
	OAT	Yes	$< 1\ \mu s$
	ARTO	Yes	$< 1\ \mu s$
ArduCopter (Attack ⑦)	OAT	Yes	$< 1\ \mu s$
	ARTO	Yes	$< 1\ \mu s$
Robotic Arm's Control Loop (Attack ⑥)	OP-DFI	Yes	$< 1\ \mu s$
	OAT	No	N/A
	ARTO	Yes	$< 1\ \mu s$
Robotic Arm's Control Loop (Attack ⑦)	OP-DFI	No	N/A
	OAT	Yes	$< 1\ \mu s$
	ARTO	Yes	$< 1\ \mu s$
PX4 (Attack ⑧)	OP-DFI	Yes	$< 1\ \mu s$
	ARTO	Yes	$< 1\ \mu s$

the time for verifier validation. For attack ④, ARI's detection time is 76.66 seconds. This delay stems primarily from the time required to produce the control-flow evidence. In ARTO, however, the local CFI violation detection takes effect immediately. This early alert enables ARTO to initiate the transmission of evidence (HBuf, IBuf, and TBuf) earlier, significantly reducing the total time for remote detection and diagnosis. The efficiency of local detection also justifies our design choice on the partial context-sensitive CFI.

Effectiveness against Data-Only Attack We compare ARTO with OP-DFI and OAT against the data-only attacks ⑥-⑧. Table 6 shows that ARTO successfully detects these attacks. In the Listing 1 program, ARTO assigns `cmd` to Equivalence Class 0 and `st` to Equivalence Class 3. Regarding attack ⑥, ARTO examines the definition table entries in Equivalence Class 3. It finds that the address of `cmd.op` is not present in this class and reports an error. Regarding attack ⑦ on different programs, ARTO consistently detects that the retrieved value is inconsistent with the value stored in the definition table. In contrast, although OAT can detect the direct modification of the critical variable in attack ⑦, it fails to identify the complex read-target manipulation in attack ⑥, demonstrating ARTO's security advantage in detecting the indirect data manipulation to coincide a critical variable with the location of another critical variable. OP-DFI's DFG-based address check can identify attack ⑥. However, OP-DFI fails to detect attack ⑦ because our experimental hardware lacks MTE support, resulting in false negatives. This observation highlights the practicality of ARTO, offering robust protection without relying on specialized hardware features such as MTE. As reported in Section 6.1, we failed to protect ArduCopter with OP-DFI. However, we successfully use OP-DFI to mitigate attack ⑧ against PX4 because this scenario focuses on a straightforward critical operation that OP-DFI can process. Meanwhile, ARTO and OP-DFI detect attack ⑧ in different ways: ARTO identifies that `number_valid_points` is outside its valid range, while OP-DFI reports the attack at the out-of-bound write.

6.3 Protection Bound on Complex CPS (RQ3)

ARTO's overhead increases with the number of critical operations. We explore the maximum number of critical operations

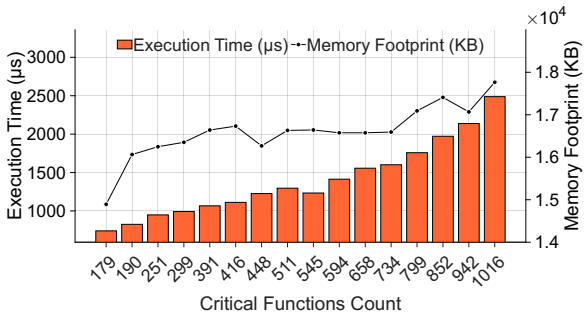


Figure 6: Protection Limit on ArduCopter

that xxHash-based ARTO can protect. ArduCopter is used in this evaluation due to its explicit maximum control loop cycles. Since critical operations vary in complexity, we use the number of critical functions as the measure of complexity. We focus on the functions in the attitude and position controllers, progressively selecting more functions into critical operations. Fig. 6 illustrates the relationship between the number of critical functions and runtime and memory overhead in ARTO. As the number of protected functions increases, the overheads increase accordingly. We observed that ARTO can operate properly until the number of critical functions approaches 1,000 under the main loop cycle limit of 2.5 ms.

6.4 Real-Time Impacts on Autopilot (RQ4)

To evaluate ARTO's impact on the real-time performance of the autopilot, we use *Extreme Value Theory* (EVT) [25] to estimate the WCET of the ArduCopter tasks listed in Fig. 4(a) under ARTO's protection, and then perform schedulability analysis using MAST [33].

Specifically, we run ArduCopter on Raspberry Pi 3B for one hour and record the execution-time samples for each task. We then apply the block maxima method [15] by dividing the execution-time samples of each task into multiple data blocks of size k, and extracting the maximum execution-time sample from each block to form a sequence of extreme-value samples. We use maximum likelihood estimation (MLE) to fit this sequence to a *Generalized Extreme Value* (GEV) distribution. Finally, based on the fitted distribution, we calculate the WCET of each task at a given confidence level α, i.e., $\text{WCET}_\alpha = \mu - \frac{\sigma}{\xi}\left[1 - (-\ln\alpha)^{-\xi}\right]$, where μ, σ, and ξ denote the location parameter, scale parameter, and shape parameter, respectively. These parameters characterize the overall location, dispersion, and tail behavior of the extreme-value distribution. We follow the method of [32] to decide k for each task. For the confidence level α, we calculate WCET_α under different values of α based on the GEV distribution fitted from the extracted extreme-value samples. We calculate the WCET estimate for each α value from the candidate confidence-level set, i.e., $\alpha \in \{0.99, 0.995, 0.999, 0.9995, 0.9999\}$. We then check each estimate against the collected execution-time samples. If no execution-time sample of a task exceeds the cor-

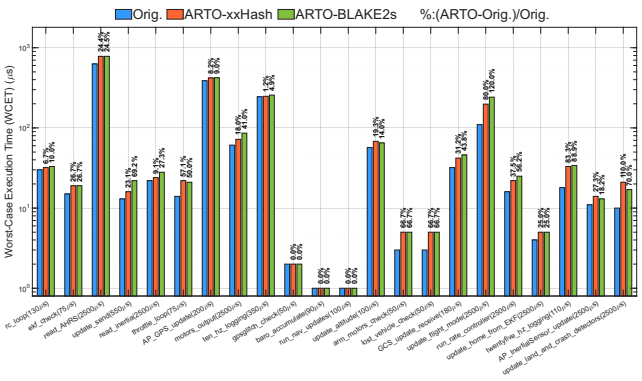


Figure 7: WCET of ARTO-Protected ArduCopter

Table 7: Schedulability of ARTO-Protected ArduCopter

	MAST Results		
	slack	CPU Total Utilization	Schedulability
ArduCopter$_{\text{Orig.}}$	37.38%	48.93%	✓
ArduCopter$_{\text{ARTO-xxHash}}$	24.12%	54.00%	✓
ArduCopter$_{\text{ARTO-BLAKE2s}}$	21.20%	56.19%	✓

responding WCET$_\alpha$, we consider the task to satisfy the coverage requirement at the current confidence level. As shown in Fig. 7, regardless of the hash algorithm used by ARTO, only the WCET of `GPS_update` exceeds the expected task execution time. However, the WCET of this task in the vanilla ArduCopter also exceeds the expected task execution time. Given that the GPS data sampling rate does not exceed 10 Hz, `GPS_update`'s high WCET still meets ArduCopter's GPS data quality requirements.

We then construct the timing-behavior model of ArduCopter based on the scheduling policy, the measured WCET results, and the task timing constraints, such as task periods and deadlines. The scheduling policy and task timing constraints are inferred from the ArduCopter documentation [12] and source code. The constructed model is then input into the MAST toolset for schedulability analysis. As shown in Table 7, both the xxHash-based and BLAKE2s-based ARTO are schedulable.

7 Conclusion

ARTO is the first approach bringing the advantages of efficient local CFI, full context-sensitive CFA, and scalable data-flow protection stronger than value-based checking to protect the critical operations of the real-time cyber-physical systems. ARTO outperforms the SOAT CFA and data-flow protection approaches for embedded systems in both runtime overhead and security.

Acknowledgments

Ruizhe Zhao, Cong Sun, Zongzhen Li, and Yunbo Wang were supported by the National Natural Science Foundation of China (Nos. 62272366, 62232013) and the Key Research and Development Program of Shaanxi Province (No. 2023-YBGY-371).

Ethical Considerations

We adopt a stakeholder-based analysis to discuss the impacts and risks of this research, the risk mitigation measures, and the justification for conducting this research.

Stakeholder Analysis This work involves three main stakeholder groups: (1) *CPS system developers* can deploy ARTO to enhance their defenses against control-flow hijacking and data-only attacks. (2) *Researchers* in software and system security can obtain reproducible results by using the methods and open-source implementations of this work. (3) *CPS system users*, e.g., drone pilots, benefit from enhanced system security. Our system reduces the risk of device loss of control caused by control-flow hijacking or data-only attacks. This research does not involve human subjects, does not collect user data, and does not process personal information; therefore, there is no ethical issue related to the lack of informed consent, privacy leakages, or misuse of user data.

Impact of the Research The publication of ARTO has both positive and negative impacts, affecting all the aforementioned stakeholders.

Positive Impacts. (1) Enhancing the security of CPS systems, particularly in drones, industrial control, and automation scenarios. Their security affects not only the devices but also user safety and environmental integrity. ARTO enhances resilience against control-flow hijacking and data-only attacks, providing CPS manufacturers with a defense paradigm towards runtime attack detection and behavior verification, and helping developers pinpoint defensive gaps and iteratively improve their systems' security design. (2) By presenting the defense pipeline and evaluation process alongside full experimental artifacts, this work establishes a reproducible research paradigm. This transparency enables independent verification and provides a reference for advancing CPS security.

Dual-use risks. This paper involves the construction and evaluation of control-flow hijacking and data-only attacks. Attackers could use the exploit code to compromise real-world CPS systems.

Mitigation To reduce the potential risks outlined above, we conducted the following mitigation strategies. We exclusively use manually constructed vulnerabilities or publicly disclosed vulnerabilities. We neither conduct research on undisclosed flaws or vulnerabilities nor release the manually constructed vulnerabilities and exploit code, thereby avoiding the introduction of new attack vectors to real-world systems or their software supply chains. Furthermore, the experimental process is strictly confined to an isolated environment. All attack and defense evaluations are completed within a controlled, isolated testing platform and are not tested against CPS systems deployed in the real world.

Justification for Research This work focuses on defense mechanisms against control-flow hijacking and data-only attacks targeting CPS systems. Conducting this research is necessary for the following reasons: (1) These attacks pose critical, yet inadequately mitigated, threats to modern CPS systems. This work proposes a defense framework that bridges this gap, extending the security boundary and enabling systematic security hardening. (2) The releases of the implementation code, evaluation scripts, and artifacts provide reproducibility and verifiability as an open research foundation.

Open Science

To facilitate transparency, reproducibility, and community engagement, ARTO's source code (including the compiler toolchain, the simulated executor, the runtime library, and the remote verifier) and the artifacts for reproducing our experimental results are publicly available at https://github.com/zz-fz-john/ARTO-Project and https://doi.org/10.5281/zenodo.20355901.

References

[1] LLVM loop terminology (and canonical forms), 2003-2026. URL: https://llvm.org/docs/LoopTerminology.html.

[2] Type metadata, 2003-2026. URL: https://llvm.org/docs/TypeMetadata.html.

[3] xxHash, 2014-2025. URL: https://xxhash.com.

[4] BLAKE2 — fast secure hashing, 2017. URL: https://www.blake2.net.

[5] Prototype for OAT: Attesting operation integrity of embedded devices, 2020. URL: https://github.com/RiS3-Lab/OAT-Project.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05*, pages 340–353. ACM, 2005.

[7] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *CCS '16*, pages 743–754. ACM, 2016.

[8] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. DIAT: data integrity attestation for resilient collaboration of autonomous systems. In *NDSS '19*. The Internet Society, 2019.

[9] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE SP '08*, pages 263–277. IEEE Computer Society, 2008.

[10] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. μRAI: securing embedded systems with return address integrity. In *NDSS '20*. The Internet Society, 2020.

[11] Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu. On bridging the gap between control flow integrity and attestation schemes. In *33rd USENIX Security Symposium*. USENIX Association, 2024.

[12] ArduPilot Dev Team. Threading – The AP_Scheduler system. https://ardupilot.org/dev/docs/learning-ardupilot-threading.html#the-ap-scheduler-system, 2024.

[13] ArduPilot Dev Team. ArduPilot: Open source autopilot software. https://ardupilot.org/, 2025.

[14] Orlando Arias, Dean Sullivan, Haoqi Shan, and Yier Jin. LAHEL: lightweight attestation hardening embedded devices using macrocells. In *HOST '20*, pages 305–315. IEEE, 2020.

[15] Jan Beirlant, Yuri Goegebeur, Johan Segers, and Jozef L Teugels. *Statistics of extremes: theory and applications*. John Wiley & Sons, 2006.

[16] Cyril Bresch. *Approaches, Strategies, and Implementations of Memory Safety Defenses in Critical and Constrained Embedded Systems*. PhD thesis, Université Grenoble Alpes, 2020.

[17] Marco Brohet and Francesco Regazzoni. A survey on thwarting memory corruption in RISC-V. *ACM Comput. Surv.*, 56(2):28:1–28:29, 2024.

[18] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, 2017.

[19] Scott A. Carr and Mathias Payer. DataShield: configurable data confidentiality and integrity. In *AsiaCCS '17*, pages 193–204. ACM, 2017.

[20] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI '06*, pages 147–160. USENIX Association, 2006.

[21] Shuo Chen, Jun Xu, and Emre Can Sezer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*. USENIX Association, 2005.

[22] Long Cheng, Ke Tian, Danfeng Daphne Yao, Lui Sha, and Raheem A. Beyah. Checking is believing: Event-aware program anomaly detection in cyber-physical systems. *IEEE Trans. Dependable Secur. Comput.*, 18(2):825–842, 2021.

[23] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *IEEE SP '17*, pages 289–303. IEEE Computer Society, 2017.

[24] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *DAC '14*, pages 133:1–133:6. ACM, 2014.

[25] Laurens De Haan and Ana Ferreira. *Extreme value theory: an introduction*. Springer, 2006.

[26] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. LiteHAX: lightweight hardware-assisted attestation of program execution. In *ICCAD '18*, page 106. ACM, 2018.

[27] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. LO-FAT: low-overhead control flow attestation in hardware. In *DAC '17*, pages 24:1–24:6. ACM, 2017.

[28] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *26th USENIX Security Symposium*, pages 131–148. USENIX Association, 2017.

[29] Embench Open Group. Embench: A modern open-source benchmark suite for embedded systems. https://www.embench.org/, 2024.

[30] Aurélien Francillon. *Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks*. PhD thesis, Grenoble Institute of Technology, France, 2009.

[31] Munir Geden and Kasper Rasmussen. TRUVIN: lightweight detection of data-oriented attacks through trusted value integrity. In *TrustCom '20*, pages 174–181. IEEE, 2020.

[32] Jeffery P. Hansen, Scott A. Hissam, and Gabriel A. Moreno. Statistical-based WCET estimation and validation. In *WCET '09*, OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2009.

[33] Michael González Harbour, J. J. Gutiérrez García, José C. Palencia Gutiérrez, and J. M. Drake Moyano. MAST: modeling and analysis suite for real time applications. In *ECRTS '01*, pages 125–134. IEEE, 2001.

[34] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *CCS '18*, pages 1470–1486. ACM, 2018.

[35] Sirine Ilahi, Adebayo Omotosho, and Christian Hammer. ShadowGuard: cryptographic shadow stack protection with XOR obfuscation and HMAC integrity. In *ISSRE '25*, pages 119–129. IEEE, 2025.

[36] Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium*, pages 195–211. USENIX Association, 2019.

[37] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *IEEE EuroS&P '19*, pages 95–110. IEEE, 2019.

[38] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS '18*. The Internet Society, 2018.

[39] Ben Niu and Gang Tan. Modular control-flow integrity. In *PLDI '14*, pages 577–587. ACM, 2014.

[40] Ben Niu and Gang Tan. Per-input control-flow integrity. In *CCS '15*, pages 914–926. ACM, 2015.

[41] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-CFA: minimalistic control-flow attestation using verified proofs of execution. In *DATE '21*, pages 641–646. IEEE, 2021.

[42] PX4 Dev Team. PX4 autopilot. https://px4.io/, 2025.

[43] Nguyen Anh Quynh. Capstone disassembly framework. https://github.com/capstone-engine/capstone, 2014.

[44] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266. ACM, 2016.

[45] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. OAT: attesting operation integrity of embedded devices. In *IEEE SP '20*, pages 1433–1449. IEEE, 2020.

[46] Xi Tan and Ziming Zhao. SHERLOC: secure and holistic control-flow violation detection on embedded systems. In *CCS '23*, pages 1332–1346. ACM, 2023.

[47] Victor van der Veen, Dennis Andriesse, Enes Göktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *CCS '15*, pages 927–940. ACM, 2015.

[48] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-flow integrity for real-time embedded systems. In *ECRTS '19*, volume 133 of *LIPIcs*, pages 2:1–2:24, 2019.

[49] Jinwen Wang, Yujie Wang, Ao Li, Yang Xiao, Ruide Zhang, Wenjing Lou, Y. Thomas Hou, and Ning Zhang. ARI: attestation of real-time mission execution integrity. In *32nd USENIX Security Symposium*, pages 2761–2778. USENIX Association, 2023.

[50] Yujie Wang, Ao Li, Jinwen Wang, Sanjoy K. Baruah, and Ning Zhang. Opportunistic data flow integrity for real-time cyber-physical systems using worst case execution time reservation. In *33rd USENIX Security Symposium*. USENIX Association, 2024.

[51] Yujie Wang, Cailani Lemieux Mack, Xi Tan, Ning Zhang, Ziming Zhao, Sanjoy K. Baruah, and Bryan C. Ward. InsectACIDE: debugger-based holistic asynchronous CFI for embedded system. In *RTAS '24*, pages 360–372. IEEE, 2024.

[52] Tianrou Xia, Hong Hu, and Dinghao Wu. DEEPTYPE: refining indirect call targets with strong multi-layer type analysis. In *33rd USENIX Security Symposium*. USENIX Association, 2024.

[53] Hao Xiang, Zehui Cheng, Jinku Li, Jianfeng Ma, and Kangjie Lu. Boosting practical control-flow integrity with complete field sensitivity and origin awareness. In *CCS '24*, pages 4524–4538. ACM, 2024.

[54] Nikita Yadav and Vinod Ganapathy. Whole-program control-flow path attestation. In *CCS '23*, pages 2680–2694. ACM, 2023.

[55] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. ATRIUM: runtime attestation resilient under memory attacks. In *ICCAD '17*, pages 384–391. IEEE, 2017.

[56] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium*, pages 1219–1236. USENIX Association, 2020.

A Supplemental Technical Details

A.1 Cyclic Structure Breaking Algorithm

For the call graph with respect to the OpCFG, Algorithm 1 calculates the *cycle appearance count* for each node and iteratively selects the highest-weighted node to break the maximum number of cycles.

Algorithm 1: Identify and Break Recursive Cycles

Input : Call Graph $G = (V, E)$ of OpCFG, Function Set F_G
Output : Acyclic Graph $G_{NoCycle}$, Set of Cycle Tail Functions S_{tail}

$S_{tail} \leftarrow \emptyset$
$Cycles \leftarrow$ GetSimpleCycles(G)
Initialize node weights $W \leftarrow \{0\}$
`// Compute node weights as their cycle appearance cnt`
foreach $cycle \in Cycles$ **do**
  foreach $node \in cycle$ **do**
    $W[node] \leftarrow W[node] + 1$
while $Cycles \neq \emptyset$ **do**
  $v_{max} \leftarrow$ argmax(W) `// Select node with max weight`
  $cycle_{target} \leftarrow$ NULL
  `// Find a cycle containing the max node`
  foreach $cycle \in Cycles$ **do**
    if $v_{max} \in cycle$ **then**
      $cycle_{target} \leftarrow cycle$
      break
  if $cycle_{target} \neq NULL$ **then**
    `// Update weights and remove cycle`
    foreach $node \in cycle_{target}$ **do**
      $W[node] \leftarrow W[node] - 1$
    $Cycles$.remove$(cycle_{target})$
  G.remove(v_{max}) `// Logically break the cycle`
  S_{tail}.add(v_{max})
  `// Re-evaluate cycles`
  $Cycles \leftarrow$ GetSimpleCycles(G)
return G, S_{tail}

A.2 Offline Measurement Generation

This section details the generation of the *valid path segment hash table* $\mathcal{D}$ and the *hash-trace map* $\mathcal{M}$, whose definitions are given in Section 3.1.2. Unlike the verifier's re-execution for checking control-flow violations, the binary-level simulated executor performs path exploration to collect offline measurements of all related sub-paths. The executor focuses exclusively on control-flow-related instructions (i.e., calls, returns, conditional and unconditional branches) to reduce complexity. The path exploration follows a depth-first search (DFS) guided by instruction semantics. A shadow stack is maintained to guide the function returns. Specifically, targets for direct transfers are statically identified from the binary, while potential targets for indirect calls are obtained from the indirect-call-resolved OpCFG of the critical operation.

The path segment $ps(v_i, v_j)$ usually has several control-flow sub-paths. The executor's DFS traverses one of these

Table 8: Example of Database Structure

Embedding Address	Header Type (32 bits)	#Hashes (32 bits)	Segment Start (32 bits)	Segment End (32 bits)
addr	NOT_END	k	v_i	v_j
addr+128	$hash_1$		$hash_2$	
...	...			
	$hash_k$		PADDING	
addr+128(n-2)	NOT_END	2	v'_i	v'_j
addr+128(n-1)	$hash'_1$		$hash'_2$	
addr+128n	END	PADDING		

sub-paths (e.g., p_s) each time. The executor maintains an *indirect-target list* $\mathcal{L}$ to store the target addresses of indirect control-flow transfers encountered during the traversal of p_s. Upon reaching the segment-ending verify point v_j, the executor obtains the accumulated hash digest (denoted as $hash_s$) of the ordered target-address sequence, e.g., $(tarAddr_1, \ldots, tarAddr_m)$, in $\mathcal{L}$ using the hash primitive for runtime measurement (Section 3.1.2). Then, the executor performs two updates.

- $\mathcal{M}$-Update: if $hash_s$ has never been encountered before, update $\mathcal{M}$ by adding $(hash_s, (tarAddr_1, \ldots, tarAddr_m))$ to $\mathcal{M}$.
- $\mathcal{D}$-Update: if $hash_s$ does not exist in the table item of $ps(v_i, v_j)$ in $\mathcal{D}$, update $\mathcal{D}$ by adding $hash_s$ to the table item of $ps(v_i, v_j)$. The number of valid hashes of $ps(v_i, v_j)$ will also be increased.

After these measurement actions at v_j, the executor clears $\mathcal{L}$ before resuming the DFS from v_j.

Finally, the overall valid path segment hash table is organized as shown in Table 8. This hash table example has two items, one for $ps(v_i, v_j)$ and the other for $ps(v'_i, v'_j)$. $ps(v_i, v_j)$ has k (an odd number) valid hash digests $\{hash_1, \ldots, hash_k\}$, while $ps(v'_i, v'_j)$ has 2 valid hash digests $\{hash'_1, hash'_2\}$. Each hash digest is in 64 bits. The complete hash table $\mathcal{D}$ is embedded into the protected runtime region of the target program for local verification. The hash-trace map $\mathcal{M}$ is offloaded to the remote verifier. This separation design minimizes overhead on the embedded device while enabling the verifier to reconstruct precise control-flow traces from the reported hashes.

In the DFS procedure, directly computing the cumulative hash for each sub-path within the path segments will be extremely expensive. We can use the *simulated path context* at the specific verify point to accelerate the DFS procedure. The path context is actually the *global shadow stack state* when the DFS reaches the verify point. Specifically, ARTO implements a *state-pruning* mechanism. For each item of $\mathcal{D}$, e.g., the one of $ps(v_i, v_j)$, ARTO's measurement action at v_j merges the current shadow stack state $\texttt{ST}(v_j)$ with the current indirect-target list $\mathcal{L}$ to obtain a *path-context hash*, i.e., $\text{Hash}(\texttt{ST}(v_j) \oplus (tarAddr_1, \ldots, tarAddr_m))$. The hash table $\mathcal{D}$ will store this unique hash value into a *path-context hash set* for the path segment $ps(v_i, v_j)$. Next time the DFS procedure reaches v_j, the measurement action at v_j will first check whether the current path-context hash value matches any entry in the path-context hash set. If so, it means the current indirect-target list has previously been recorded under the same path context, the executor can backtrack to prune the redundant paths; otherwise, the exploration continues. The path-context hash set is maintained only during the DFS procedure and will not be embedded in the final binary.

A.3 Prover-Side Context Sensitivity

Although ARTO records only the target addresses of indirect transfers at runtime, and the path-segmentation mechanism may degrade partial context sensitivity into context insensitivity in some cases (as shown in Listing 2), ARTO's CFI still preserves partial context sensitivity in most scenarios. For example, based on the choices of verify points, there can be different sub-paths (e.g., $v_1 \to v_2$ and $v_3 \to v_4$) passing through the same return site *ret*. The valid return edge on $v_1 \to v_2$ is (ret, tar_1), while the valid return edge on $v_3 \to v_4$ is (ret, tar_2). If the cumulated hash of the indirect targets along $v_1 \to ret$ is different from the cumulated hash along $v_3 \to ret$, then any attack attempt to hijack a runtime (ret, tar_1) to (ret, tar_2) will be blocked by ARTO's local CFI. Because there is no hash digest for $v_1 \to ret \to v_4$ in the hash database. Therefore, our local CFI can still provide partial context-sensitive protection. The situation becomes difficult when the attacker-targeted return site *ret* issues the first indirect transfer of different sub-paths, our CFI reduces to context-insensitive in this case. Another case is $v_1 \to ret$ and $v_3 \to ret$ share the same cumulated hash value (essentially the case in Listing 2), also causing insufficient context information for our local CFI. Given that our verify point determination selects many leaf functions as verify points, we argue that the sub-path distribution benefits from this partial context sensitivity and is definitely more strict than the context-insensitive CFI in real-world programs.

A.4 Fixpoint for Critical Variable Inclusion

From the initial critical variable set S_{crit}, ARTO employs a *fixed-point iteration* algorithm to identify the complete critical variable set in the critical operation. Before the iteration, ARTO implements a field-sensitive context-insensitive Andersen pointer analysis to obtain the global points-to relations. The algorithm takes the global points-to information as input, iteratively expands S_{crit} by simultaneously applying two propagation rules until the set stabilizes:

- Backward dependency propagation: By traversing the *use-def* chains at the LLVM-IR level, any variable that operates as an operand to define a variable currently in S_{crit} is identified as an upstream critical variable and added to the set.
- Points-to expansion: Simultaneously, we query the points-to relations to identify any pointer variable that may refer to the same memory region as a variable in S_{crit}. These pointer variables are deemed critical and added to the set.

Table 9: SFI-Related Instrumentation of ARTO

Type	Orig. Instr.	Instrumented Instr.

indirect call of critical function	`blx rx`	`mov` r_{rsv}`, rx`
		`bl collect_indirect_call`
		`mov` r_{rsv}`, 0`
		`bfi` r_{rsv}`, rx, #0, n`
		`blx` r_{rsv}

return of critical function	`pop pc`	`pop lr`
		`mov` r_{rsv}`, lr`
		`push lr`
		`bl collect_ret`
		`pop lr`
		`mov` r_{rsv}`, 0`
		`bfi` r_{rsv}`, lr, #0, n`
		`bx` r_{rsv}

indirect call of non-critical function	`blx rx`	`mov` r_{rsv}`, 0`
		`bfi` r_{rsv}`, rx, #0, n`
		`blx` r_{rsv}

return of non-critical function	`pop pc`	`pop lr`
		`mov` r_{rsv}`, 0`
		`bfi` r_{rsv}`, lr, #0, n`
		`bx` r_{rsv}

data write	`str ry, [rx]`	`str ry, [rx]`
		`push lr`
		`mov` r_{rsv}`, rx`
		`bl str_check`†
		`pop lr`

† `str_check` checks if the address in r_{rsv} is within the protected region. If so, aborting; otherwise, restoring registers and returning safely.

This iterative process terminates when the *fixed point* is reached (i.e., no new variables are added to the critical variable set during the complete iteration), ensuring that the transitive closure of all potential attack surfaces, whether through direct data flow or indirect memory tampering, is fully captured.

A.5 Instrumentation for SFI-based Protection

The specific assembly-level code transformations for ARTO's SFI-based protection are shown in Table 9. ARTO performs an intra-basic-block analysis to identify the candidate memory writes under instrumentation. Specifically, our analysis traverses the basic block while maintaining a list of candidate store instructions `IV`, which is initialized to empty. Our algorithm iterates on the instruction sequence as follows: 1) When a store instruction employing indirect addressing (e.g., using base register R_x) is encountered, we add it to `IV`. 2) If a subsequent store instruction uses the same base register R_x, we replace the corresponding old instruction in `IV` with this instruction. 3) If an instruction modifies a register that serves as the base for a store instruction in `IV`, we insert a call to the address-check function immediately after the store instruction and remove the store instruction from `IV`. 4) At the end of traversal, ARTO inserts an address-check function call after each store instruction remaining in `IV`.

To implement the SFI-based protection, we implement an LLVM backend pass. This backend pass operates on the result of the control-flow-related backend pass (Section 4). It first performs basic-block-level analysis to select candidate store instructions. Then, it inserts `bfi` instruction before indirect transfer instructions and inserts the store-target check mechanism after the selected store instructions.

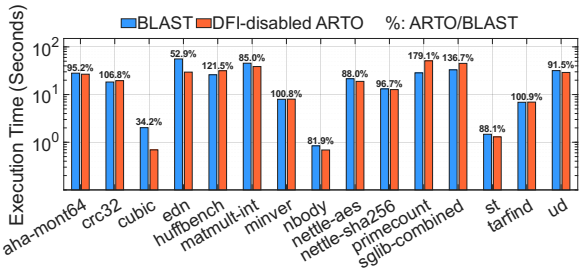
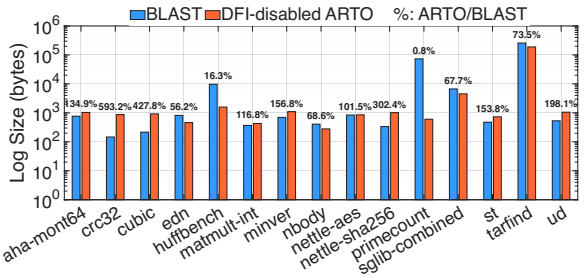
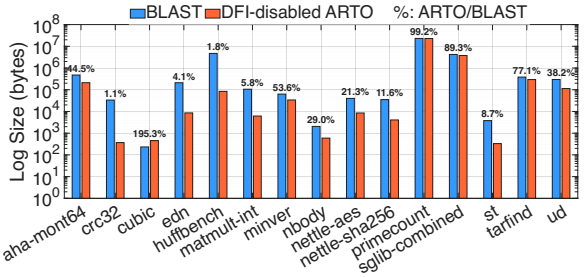


Figure 8: Execution Time Comparison with BLAST



(a) WPP-Compressed Log Size



(b) Bzip2-Compressed Log Size

Figure 9: Log Size Comparison with BLAST

B Supplemental Evaluation Results

We compare ARTO with BLAST [54]. Without BLAST's implementation, we use the data reported in the paper [54] as a baseline. We evaluated ARTO on the Embench suite with full-program protection. To ensure a fair comparison, we overclocked the Raspberry Pi 3B to 1.4 GHz and turned off our data-flow protection. This setting excludes DFI overhead and focuses on the overheads of control-flow protection. The results in Fig. 8 show that ARTO achieves execution times comparable to BLAST by filtering out specific conditional branches from control-flow events. In Fig. 9, we compare the log of both approaches in grammar-compressed (WPP) size and bzip2-compressed size. Generally, ARTO generates smaller bzip2-compressed logs than BLAST, considering we compress multiple indirect control flow events into a single hash digest and encode conditional jumps within the byte stream. In contrast, BLAST produces smaller WPP-compressed logs because the Ball-Larus path numbers used in BLAST's log are more suitable for WPP construction than ARTO's 0/1-sequence for the selected conditional branches.