

西安電子科技大學
XIDIAN UNIVERSITY

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

Ruizhe Zhao
Xidian University
ruizhezhao@stu.xidian.edu.cn

Cong Sun✉
Xidian University
suncong@xidian.edu.cn

Zongzhen Li
Xidian University

Tiantian Wang
Northeastern University
wang.tianti@northeastern.edu

Yunbo Wang
Xidian University

USENIX Security 2026

Outline

1 Background and motivation

2 Design

3 Evaluation

Background and Motivation

- **Widespread adoption of real-time Cyber-Physical Systems.**

  e.g. Drones, robotic vehicles, autonomous systems, …



- **Control-flow hijacking can trigger unsafe operation immediately.**

  CVE-2025-15150 on PX4 can perform control- flow hijack

- **Data-only attacks manipulate critical decisions without involving illegal control flow.**

  CVE-2025-5640 on PX4 can trigger out-range write

- **Protection of execution integrity of CPS**

Background – Defense

Control flow integrity (CFI)

- Verify indirect targets against the control-flow graph at runtime locally.
- **Context sensitive CFI** introduce **high overhead.**
- RECFISH(ECRTS'2019)

Control flow attestation (CFA)

- Prover records and sends the control flow evidence, remote verifier verifies the evidence post-hoc.
- **Lack of timely attack prevention.**
- ARI(USENIX'2023), OAT(S&P'2020)

Data flow integrity (DFI)

- Checks whether runtime data accesses follow the valid def-use relations defined by the data-flow graph.
- **Significant overhead.**
- DFI (OSDI'2006), OAT(S&P'2020), OP-DFI(USENIX'2024)

Landscape of SOTA control-flow and data-flow defenses for CPS (●: 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.

Motivation

- Widely deployable
- Timely protection
- High precision
- Low overhead

Threat Model and Scope

Attacker Capabilities :

- Knowledge of the memory layout
- Capabilities of arbitrary memory read/write
- Capabilities of hijack the control flow
- Capabilities of manipulate critical variables

Assumption:

- DEP (Data Execution Prevention) for preventing code injection
- Secure timer for periodic self-checking.

Scope:

- Control-flow hijacking and data manipulation attack.
- Denial-of-service attacks, physical tampering, and side-channel attacks are out of scope

Outline

1 Background and motivation

2 Design

3 Evaluation

Design of ARTO

Managing control-flow protection overhead

Local context sensitive produce **high overhead**	**Partial context-sensitive verification at runtime and full context-sensitive at remote verifier**
Recording every indirect transfer incurs **storage overhead**	Encode the execution path as a **hash digest**, verify only at designated verification points (verify points)
Path explosion	Break loops and recursions

Managing data-flow protection overhead

DFG checking incur high overhead	Group may-aliased variables into equivalence sets, check set-membership at runtime instead of full DFG traversal

Self-protection

Protecting ARTO's code and data from untrusted code	Software fault isolation (SFI)

Control Flow Integrity

Enforce partial context-sensitivity without shadow stack

- Selection of ‘verify points’

Objectives:

- **Preventing path explosion.**
- **Optimizing verification frequency.**

Strategies:

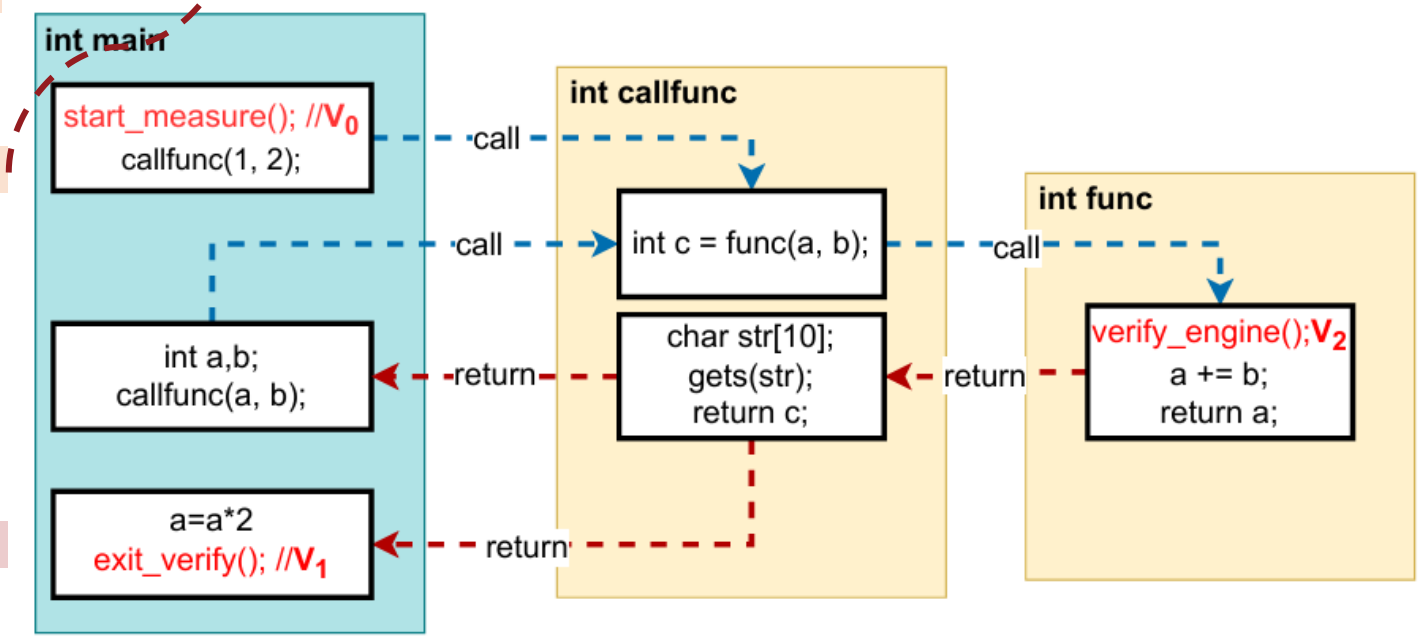
- Entry of each loop’s latch basic block
- Recursive tail function entry/exit
- Selected leaf function entry (via clustering)

Control Flow Integrity

```
f0    int main() {
v0      start_measure();                    // Operation Start
        callfunc(1, 2);
tar1    int a,b;
        callfunc(a, b);
tar2    a=a*2
v1      exit_verify();                      //operation end
        return 0;
      }
f1    int callfunc(int a, int b) {
        int c = func(a, b);
tar3    char str[10];
        gets(str);                          // buggy
ret1    return c;
      }
f2    int func(int a, int b) {              // Leaf Function
v2      verify_engine();
        a += b;
ret2    return a;
      }
```

entry & exit point of critical operation

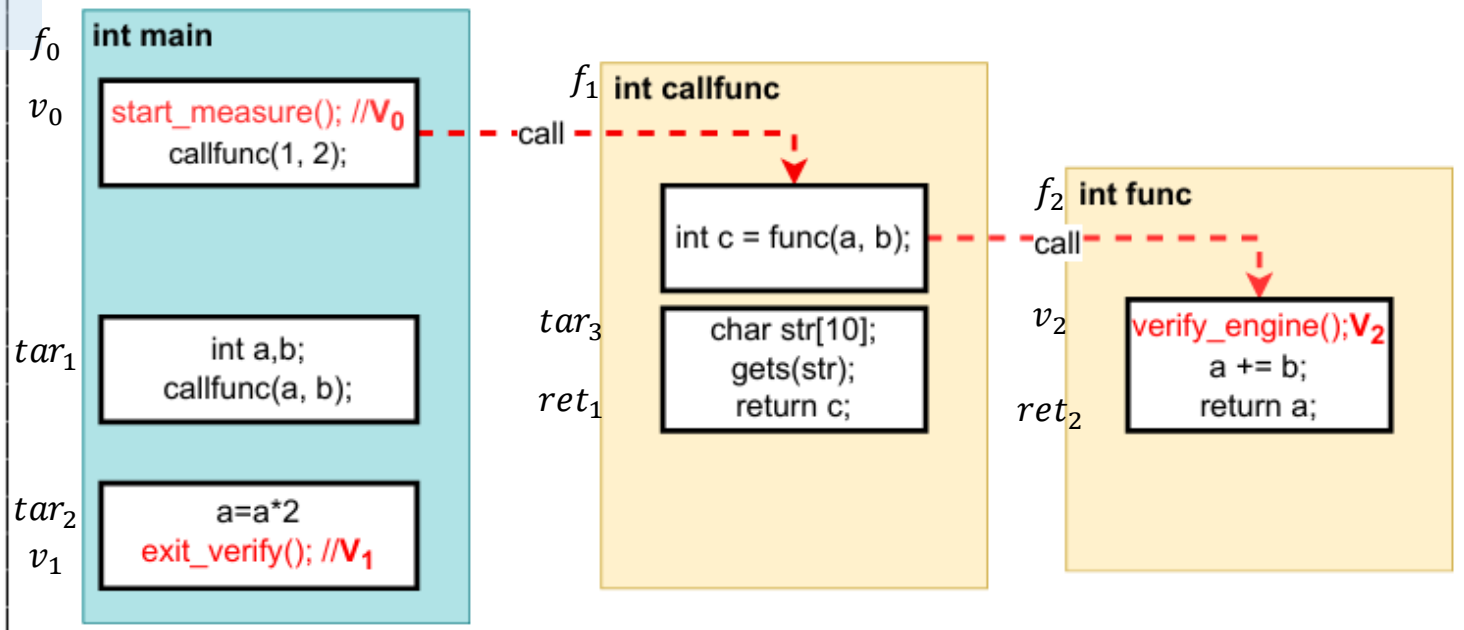
call verify engine



Original control flow graph

Control Flow Integrity

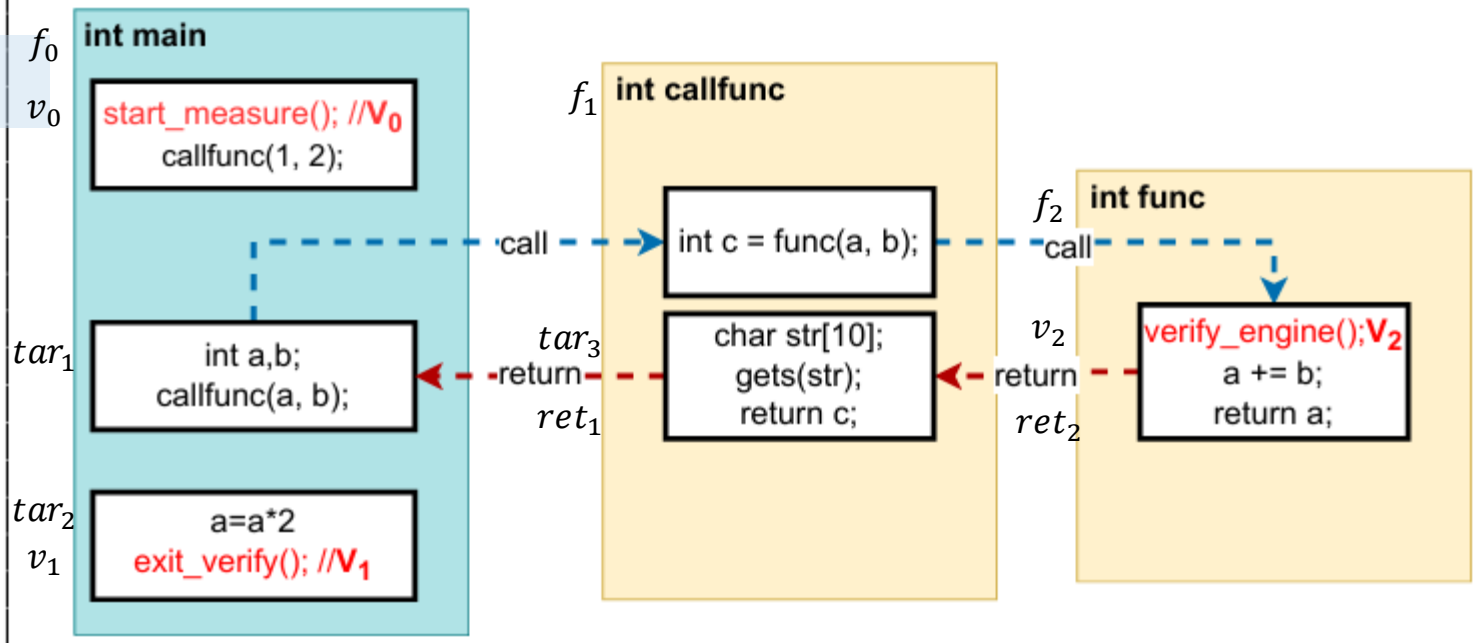
```
f0    int main() {
v0      start_measure();                    // Operation Start
        callfunc(1, 2);
tar1    int a,b;
        callfunc(a, b);
tar2    a=a*2
v1      exit_verify();                      //operation end
        return 0;
      }
f1    int callfunc(int a, int b) {
        int c = func(a, b);
tar3    char str[10];
        gets(str);                          // buggy
ret1    return c;
      }
f2    int func(int a, int b) {              // Leaf Function
v2      verify_engine();
        a += b;
ret2    return a;
      }
```



Path segment: $V_0 \rightarrow V_2$

Control Flow Integrity

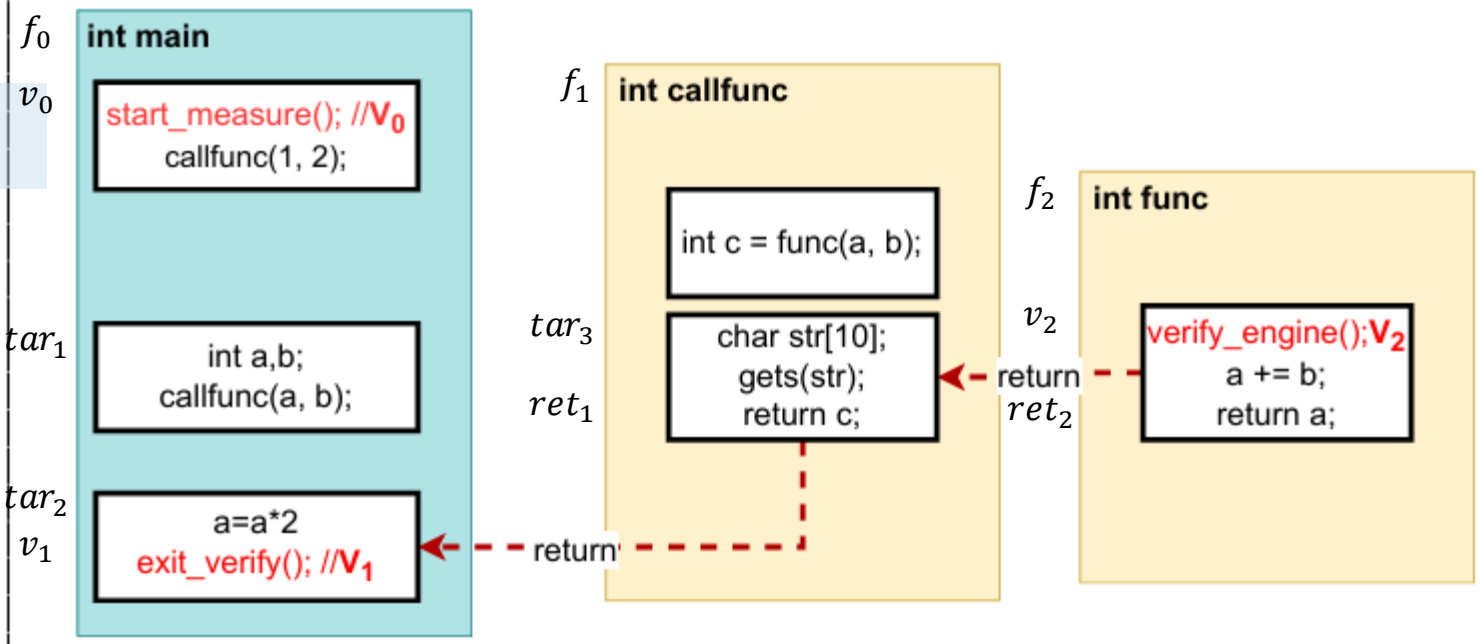
```
f0   int main() {
v0     start_measure();                    // Operation Start
       callfunc(1, 2);
tar1   int a,b;
       callfunc(a, b);
tar2   a=a*2
v1     exit_verify();                      //operation end
       return 0;
     }
f1   int callfunc(int a, int b) {
       int c = func(a, b);
tar3   char str[10];
       gets(str);                          // buggy
ret1   return c;
     }
f2   int func(int a, int b) {              // Leaf Function
v2     verify_engine();
       a += b;
ret2   return a;
     }
```



Path segment: $V_2 \rightarrow V_2$

Control Flow Integrity

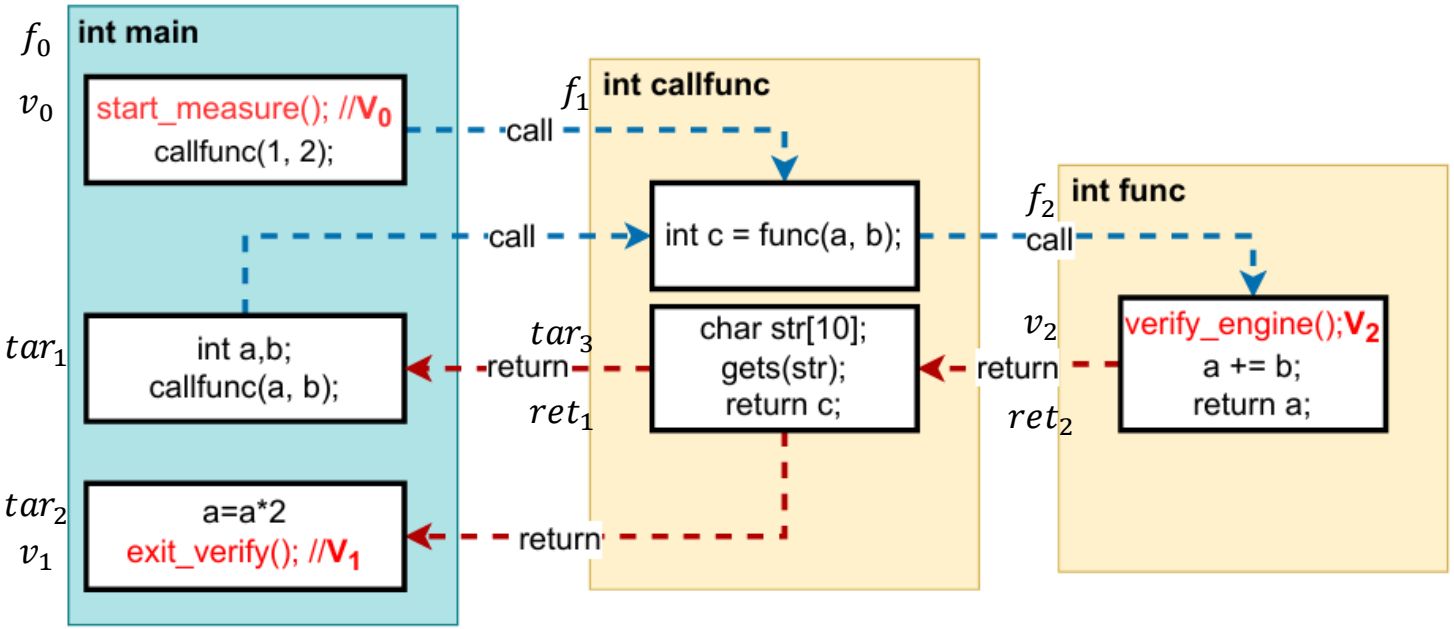
```
f0   int main() {
v0     start_measure();                    // Operation Start
       callfunc(1, 2);
tar1   int a,b;
       callfunc(a, b);
tar2   a=a*2
v1     exit_verify();                      //operation end
       return 0;
     }
f1   int callfunc(int a, int b) {
       int c = func(a, b);
tar3   char str[10];
       gets(str);                          // buggy
ret1   return c;
     }
f2   int func(int a, int b) {              // Leaf Function
v2     verify_engine();
       a += b;
ret2   return a;
     }
```



Path segment: $V_2 \rightarrow V_1$

Control Flow Integrity

Generating Valid Path-Seg Hash Table (for local CFI) using simulated execution



Valid Path-Seg Hash Table:

- $D : V \times V \mapsto H^*$

  Key → path segments

  Value → hash digests

- Used **locally, on the prover**, for the partial context-sensitive CFI check.

Path Segment	Target Sequence	Valid Hash
$ps(v_0, v_2)$	$\varnothing$	$\varnothing$
$ps(v_2, v_2)$	(tar_3, tar_1)	h_1
$ps(v_2, v_1)$	(tar_3, tar_2)	h_2

key value

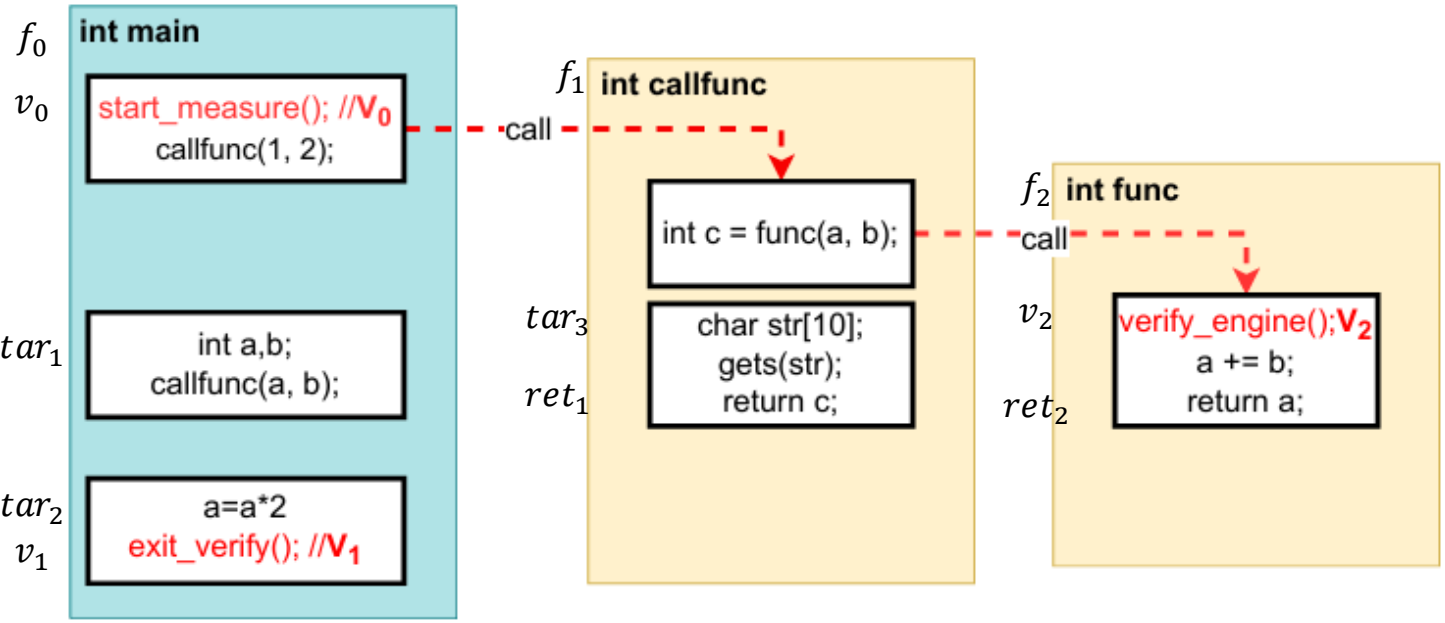
$$h_1 = \text{Hash}(\text{Hash}(tar_3) \oplus tar_1),$$

$$h_2 = \text{Hash}(\text{Hash}(tar_3) \oplus tar_2).$$

$\oplus$ denotes the concatenation operation

Control Flow Integrity

Generating Valid Path-Seg Hash Table (for local CFI) using simulated execution



Valid Path-Seg Hash Table:

- $D : V \times V \mapsto H^*$

Key → path segments
Value → hash digests

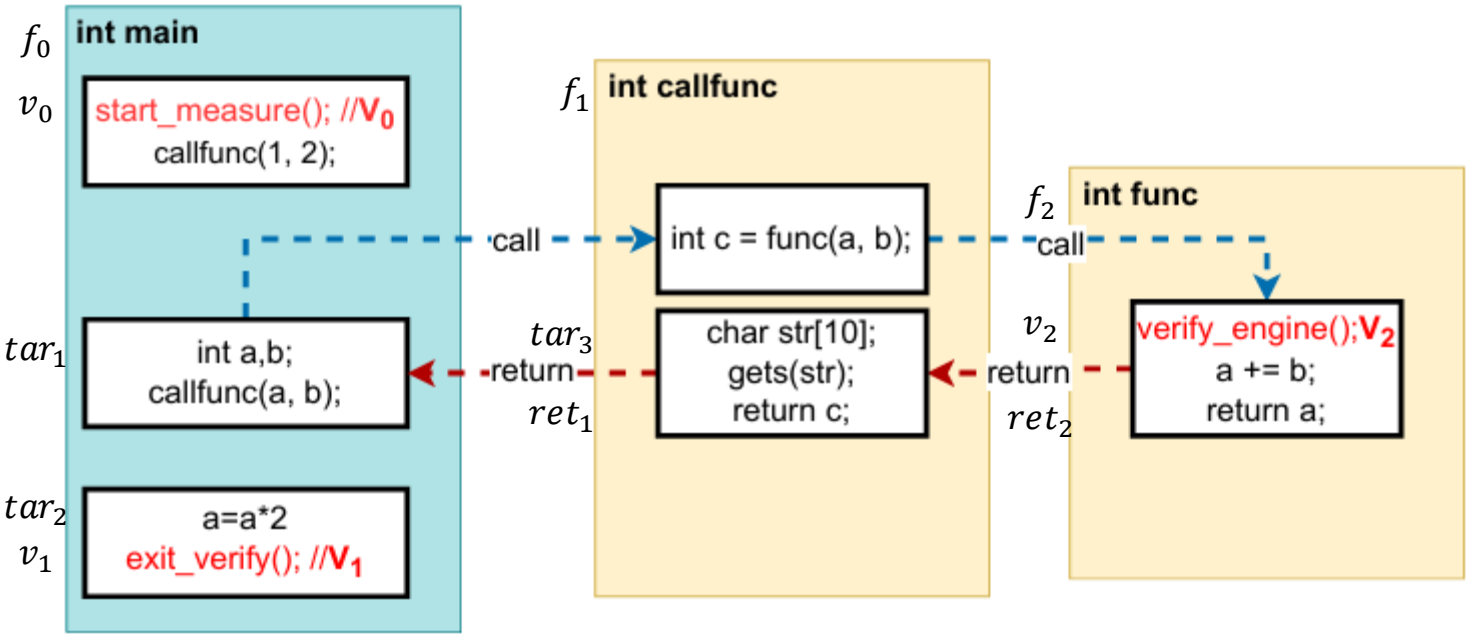
- Used **locally, on the prover**, for the partial context-sensitive CFI check.

Path Segment	Target Sequence	Valid Hash
$ps(v_0, v_2)$	$\varnothing$	$\varnothing$

Path segment: $V_0 \rightarrow V_2$

Control Flow Integrity

Generating Valid Path-Seg Hash Table (for local CFI) using simulated execution



Valid Path-Seg Hash Table:

- $D : V \times V \mapsto H^*$

Key → path segments
Value → hash digests

- Used **locally, on the prover**, for the partial context-sensitive CFI check.

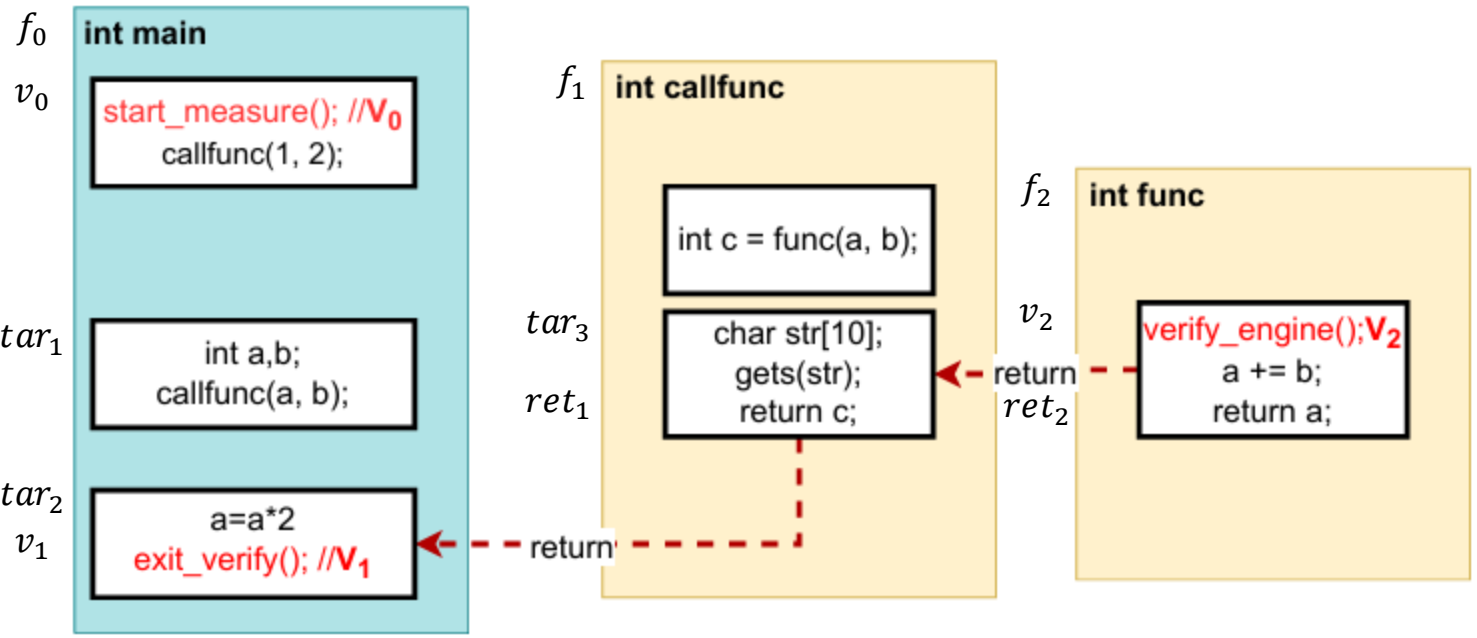
Path Segment	Target Sequence	Valid Hash
$ps(v_0, v_2)$	$\varnothing$	$\varnothing$
$ps(v_2, v_2)$	(tar_3, tar_1)	h_1

Path segment: $V_2 \rightarrow V_2$

$$h_1 = \text{Hash}(\text{Hash}(tar_3) \oplus tar_1)$$

Control Flow Integrity

Generating Valid Path-Seg Hash Table (for local CFI) using simulated execution



Path segment: $V_2 \rightarrow V_1$

Valid Path-Seg Hash Table:

- $D : V \times V \mapsto H^*$

Key → path segments

Value → hash digests

- Used **locally, on the prover**, for the partial context-sensitive CFI check.

Path Segment	Target Sequence	Valid Hash
$ps(v_0, v_2)$	$\varnothing$	$\varnothing$
$ps(v_2, v_2)$	(tar_3, tar_1)	h_1
$ps(v_2, v_1)$	(tar_3, tar_2)	h_2

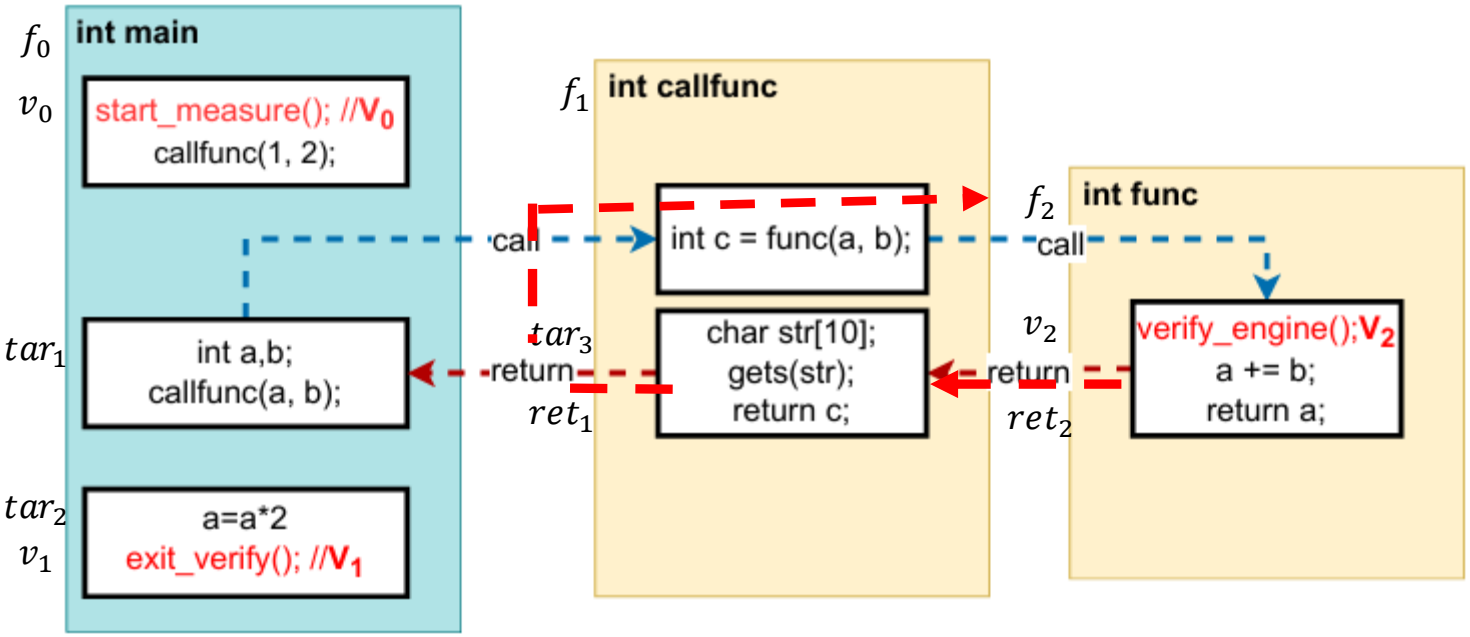
$$h_1 = \text{Hash}(\text{Hash}(tar_3) \oplus tar_1),$$

$$h_2 = \text{Hash}(\text{Hash}(tar_3) \oplus tar_2).$$

$\oplus$ denotes the concatenation operation

Control Flow Integrity

Example of attack



Path segment: $V_2 \rightarrow V_2$

Legitimate control flow is ret_2 - tar_3 - ret_1 - tar_1

Legitimate target return address sequence of path segment (v_2, v_2) is (tar_3, tar_1)

The attacker redirect ret_1 to f_2

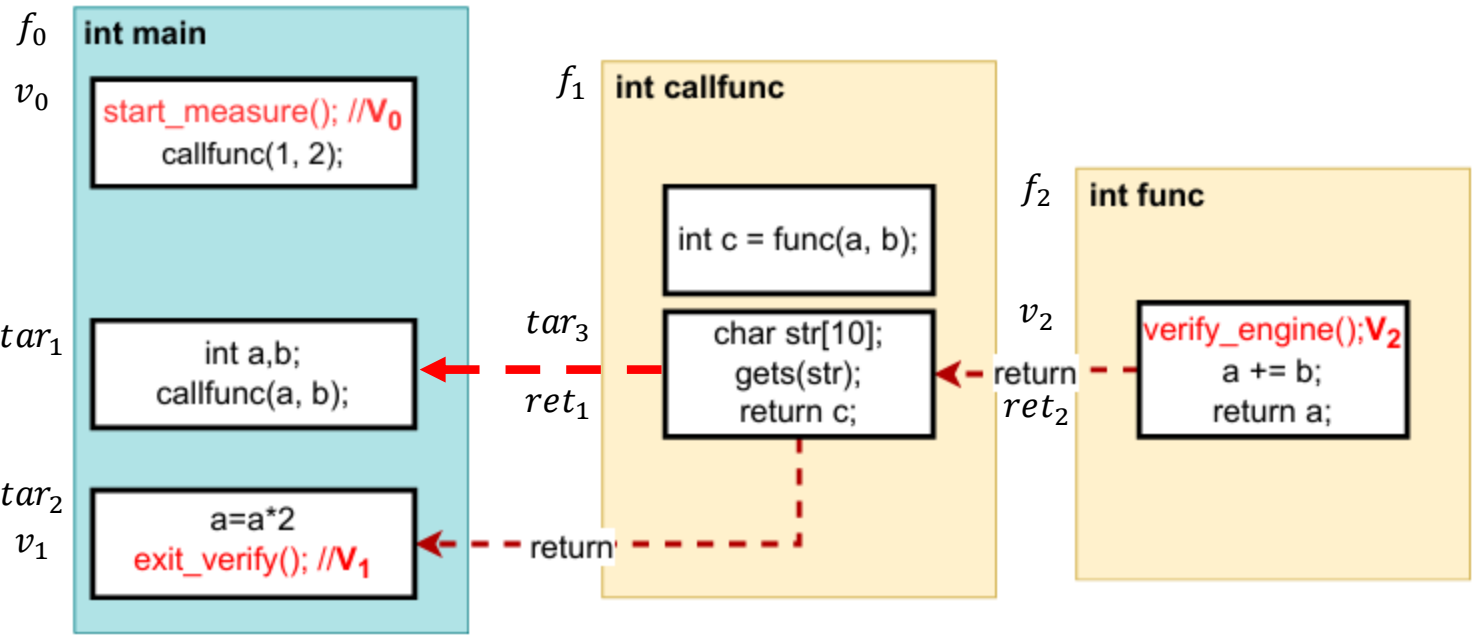
The control flow changes to ret_2 - tar_3 - ret_1 - f_2

The target return address sequence is (tar_3, f_2), and it **is NOT IN** Valid Path-Seg Hash Table

Path Segment	Target Sequence	Valid Hash
$ps(v_0, v_2)$	$\varnothing$	$\varnothing$
$ps(v_2, v_2)$	(tar_3, tar_1)	h_1
$ps(v_2, v_1)$	(tar_3, tar_2)	h_2

Control Flow Attestation

Example of attack



Path segment: $V_2 \rightarrow V_1$

Legitimate control flow is ret_2 - tar_3 - ret_1 - tar_2

Legitimate target return address sequence of path segment (v_2, v_1) is (tar_3, tar_2)

The attacker redirect ret_1 to tar_1

The control flow changes to ret_2 - tar_3 - ret_1 - tar_1

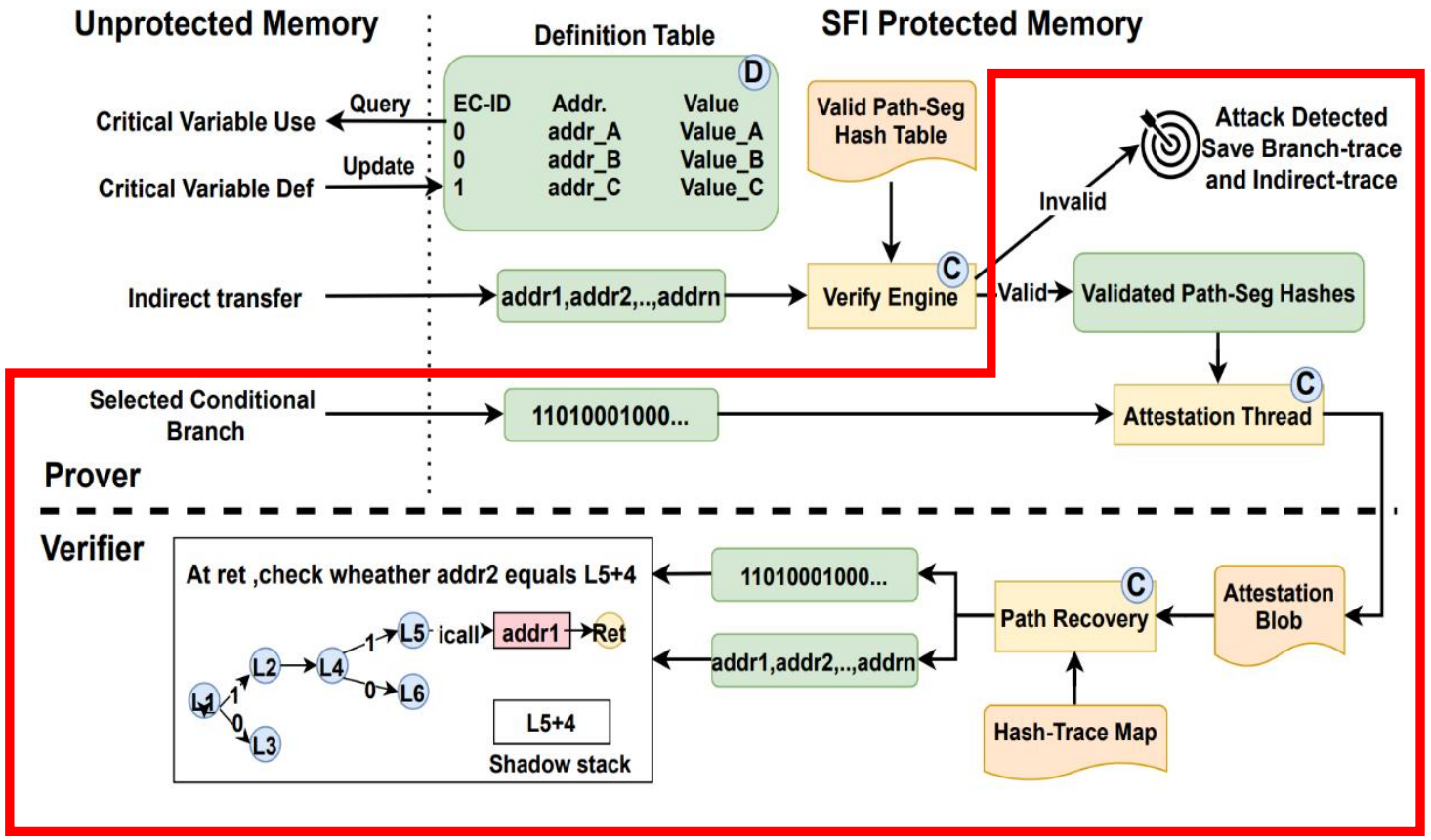
The target return address sequence is (tar_3, tar_1), and it **is IN** Valid Path-Seg Hash Table.

Path Segment	Target Sequence	Valid Hash
$ps(v_0, v_2)$	$\varnothing$	$\varnothing$
$ps(v_2, v_2)$	(tar_3, tar_1)	h_1
$ps(v_2, v_1)$	(tar_3, tar_2)	h_2

CFI fails to detect the attack.

Control Flow Attestation

Remote path recovery and global context-sensitive re-execution



- Prover sends **HBuf** + **TBuf** (signed, with nonce)
- Verifier recovers targets from HBuf via **hash-trace map**
- **Re-executes OpCFG** using branch bits (forks), binary (direct calls), and resolved OpCFG (indirect calls)
- **Shadow stack** check on each return → detects hijacking

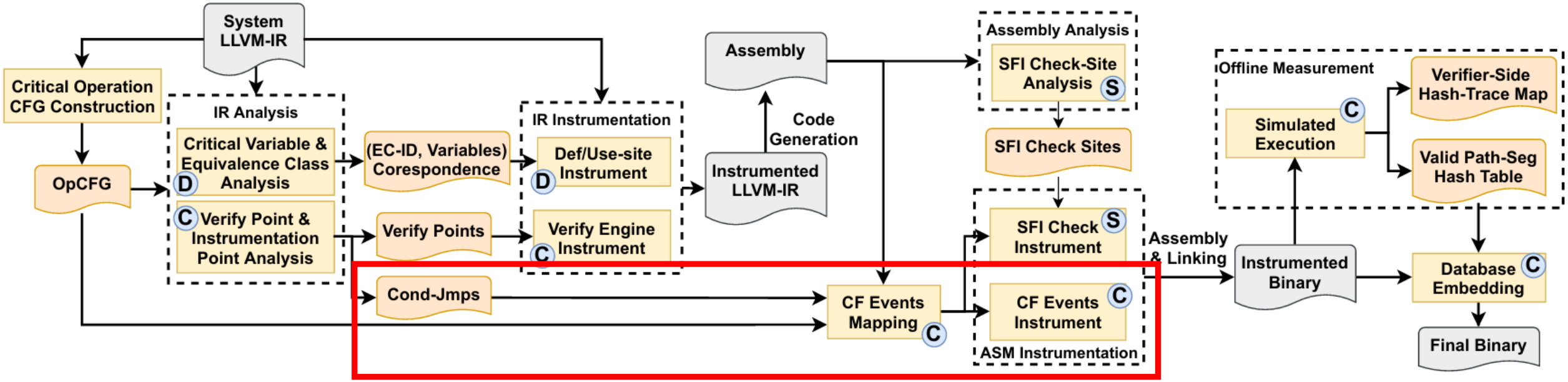
Control Flow Attestation

Control-flow Recording Function Insertion

- Selected Conditional branch , Indirect call , Return

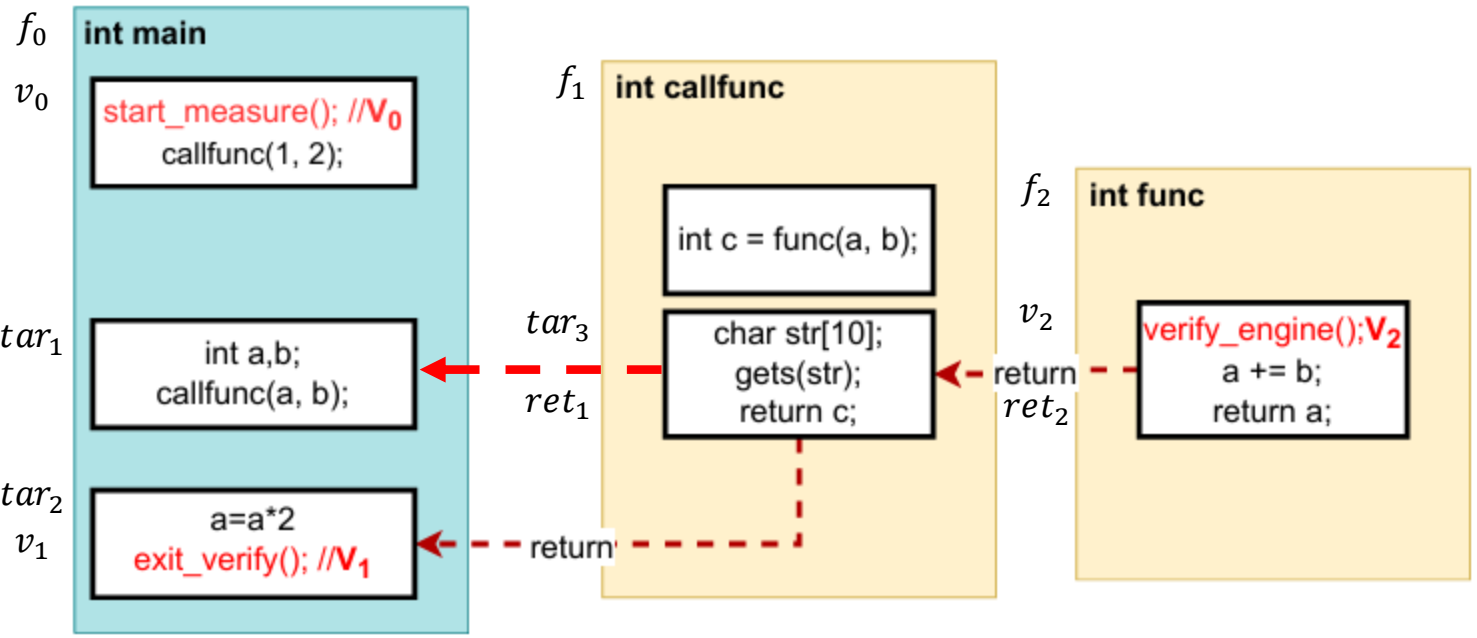
Conditional Branch Selection

- Conditional branch whose target paths contain neither function calls nor function-pointer assignments are excluded.



Control Flow Attestation

Example of attack



Path segment: $V_2 \rightarrow V_1$

Legitimate control flow is ret_2 - tar_3 - ret_1 - tar_2

Legitimate target return address sequence of path segment (v_2, v_1) is (tar_3, tar_2)

The attacker redirect ret_1 to tar_1

The control flow changes to ret_2 - tar_3 - ret_1 - tar_1

The target return address sequence recovered by verifier is (tar_3, tar_1).

The verifier's re-execution expects (tar_3, tar_2) — **mismatch reveals the hijack**

Critical Variable Protection

Critical Variable Identification

- Variables annotated by developers or variables in branch conditions
- Expand the set through the data dependencies or points-to relations

```
// Simplified control loop for robotic arms
void main_looper() {
    int st = 0;
    cmd_t cmd;                                          Annotated variable: cmd
    int (*op_func)(int, char*);
    while(1) {
        if (st = read_command(&cmd)) {
            get_input();
            if (st) {                                   Conditional variable: st
                op_func = get_op_func(cmd->op);
                (*op_func)(cmd->p_size, cmd->param);
            }
        } ...
    } ...
}
```

Critical Variable Protection

Equivalence-Class Analysis

- Groups may-aliased critical variables into the same equivalence class
- Assign a unique ID to each equivalence class.

```
// Simplified control loop for robotic arms
void main_looper() {
    int st = 0;
    cmd_t cmd;
    int (*op_func)(int, char*);
    while(1) {
        if (st = read_command(&cmd)) {
            get_input();
            if (st) {
                op_func = get_op_func(cmd->op);
                (*op_func)(cmd->p_size, cmd->param);
            }
        } ...
    } ...
}
```

Critical Variable	Equivalence class ID
cmd	0
st	1

Critical Variable Protection

```
// Simplified control loop for robotic arms
void main_looper() {
    def_record(addr_st, value=0, EC-ID=1);
    int st = 0;                                          //st's def site
    cmd_t cmd;
    int (*op_func)(int, char*);
    while(1) {
        def_record(addr_st, value=1, EC-ID=1);
        if (st = read_command(&cmd))                     // assume return 1
        {
            get_input();
            if (st)                                      //st's use site
            use_check(addr_st, value_st, EC-ID=1);
            {
                op_func = get_op_func(cmd->op);

    use_check(addr_cmd->op, value_cmd->op, EC-ID=0);
                (*op_func)(cmd->p_size, cmd->param);

    use_check(addr_cmd->p_size, value_cmd->p_size, EC-ID=0);

    use_check(addr_cmd->param, value_cmd->param, EC-ID=0);
            }
        } ...
    } ...
}
```

Insert **Recording Operation** before def-site

Record and update address, value and Equivalence Class ID (EC-ID) into the *definition table* (DT) within the protected memory region.

Insert **Checking Operation** after use-site

Check if the address belongs to EC-ID's valid set and if the loaded value match the recorded value in the *definition table* (DT) .

Critical Variable Protection

```
// Simplified control loop for robotic arms
void main_looper() {
  def_record(addr_st, value=0, EC-ID=1);
  int st = 0;                                          //st's def site
  cmd_t cmd;
  int (*op_func)(int, char*);
  while(1) {
    def_record(addr_st, value=1, EC-ID=1);
    if (st = read_command(&cmd))                       // assume return 1
    {
      get_input();
      if (st)                                          //st's use site
      use_check(addr_st, value_st, EC-ID=1);
      {
        op_func = get_op_func(cmd->op);

  use_check(addr_cmd->op, value_cmd->op, EC-ID=0);
        (*op_func)(cmd->p_size, cmd->param);

  use_check(addr_cmd->p_size, value_cmd->p_size, EC-ID=0);

  use_check(addr_cmd->param, value_cmd->param, EC-ID=0);
      }
    } ...
  } ...
}
```

update

st

Definition Table

EC-ID	Address	Value
1	$addr_{st}$	0

Critical Variable Protection

```
// Simplified control loop for robotic arms
void main_looper() {
   def_record(addr_st, value=0, EC-ID=1);
   int st = 0;                                          //st's def site
   cmd_t cmd;
   int (*op_func)(int, char*);
   while(1) {
      def_record(addr_st, value=1, EC-ID=1);
      if (st = read_command(&cmd))                      // assume return 1
      {
          get_input();
          if (st)                                       //st's use site
          use_check(addr_st, value_st, EC-ID=1);
          {
              op_func = get_op_func(cmd->op);

    use_check(addr_cmd->op, value_cmd->op, EC-ID=0);
              (*op_func)(cmd->p_size, cmd->param);

    use_check(addr_cmd->p_size, value_cmd->p_size, EC-ID=0);

    use_check(addr_cmd->param, value_cmd->param, EC-ID=0);
          }
      } ...
   } ...
}
```

update

Definition Table

	EC-ID	Address	Value
st	1	$addr_{st}$	1

Critical Variable Protection

```
// Simplified control loop for robotic arms
void main_looper() {
    def_record(addr_st, value=0, EC-ID=1);
    int st = 0;                                          //st's def site
    cmd_t cmd;
    int (*op_func)(int, char*);
    while(1) {
        def_record(addr_st, value=1, EC-ID=1);
        if (st = read_command(&cmd))                     // assume return 1
        {
            get_input();
            if (st)                                      //st's use site
            use_check(addr_st, value_st, EC-ID=1);
            {
                op_func = get_op_func(cmd->op);

    use_check(addr_cmd->op, value_cmd->op, EC-ID=0);
                (*op_func)(cmd->p_size, cmd->param);

    use_check(addr_cmd->p_size, value_cmd->p_size, EC-ID=0);

    use_check(addr_cmd->param, value_cmd->param, EC-ID=0);
            }
        } ...
    } ...
}
```

If $addr_{st}$ in EC 1:

- Yes :
  - If $Value_{st}$ =1
    - Yes: continue execution
    - No : data corruption
- No: data corruption

check

Definition Table

	EC-ID	Address	Value
st	1	$addr_{st}$	1

Critical Variable Protection

```
// Simplified control loop for robotic arms
void main_looper() {
    def_record(addr_st, value=0, EC-ID=1);
    int st = 0;                                          //st's def site
    cmd_t cmd;
    int (*op_func)(int, char*);
    while(1) {
        def_record(addr_st, value=1, EC-ID=1);
        if (st = read_command(&cmd))                     // assume return 1
        {
            get_input();
            if (st)                                      //st's use site
            use_check(addr_st, value_st, EC-ID=1);
            {
                op_func = get_op_func(cmd->op);

    use_check(addr_cmd->op, value_cmd->op, EC-ID=0);
                (*op_func)(cmd->p_size, cmd->param);

    use_check(addr_cmd->p_size, value_cmd->p_size, EC-ID=0);

    use_check(addr_cmd->param, value_cmd->param, EC-ID=0);
            }
        } ...
    } ...
}
```

If attacker modifies $value_{st}$ from 1 to 0

If $addr_{st}$ in EC 1:
- Yes :
  - If $Value_{st}$ =1
    - Yes: continue execution
    - No : data corruption
- No: data corruption

Definition Table

	EC-ID	Address	Value
st	1	$addr_{st}$	1

Critical Variable Protection

```
// Simplified control loop for robotic arms
void main_looper() {
    def_record(addr_st, value=0, EC-ID=1);
    int st = 0;                                          //st's def site
    cmd_t cmd;                                                    s
    int (*op_func)(int, char*);
    while(1) {
        def_record(addr_st, value=1, EC-ID=1);
        if (st = read_command(&cmd))                  // assume return 1
        {
            get_input();
            if (st)                                          //st's use site
            use_check(addr_st, value_st, EC-ID=1);
            {
                op_func = get_op_func(cmd->op);

    use_check(addr_cmd->op, value_cmd->op, EC-ID=0);
                (*op_func)(cmd->p_size, cmd->param);

    use_check(addr_cmd->p_size, value_cmd->p_size, EC-ID=0);

    use_check(addr_cmd->param, value_cmd->param, EC-ID=0);
            }
        } ...
    } ...
}
```

If attacker modifies $addr_{st}$ to $addr_{cmd}$

If $addr_{st}$ in EC 1:
- Yes :
  - If $Value_{st}$ =1
    - Yes: continue execution
    - No : data corruption
- No: data corruption

Definition Table

	EC-ID	Address	Value
st	1	$addr_{st}$	1
cmd	0	$addr_{cmd}$	5

Software Fault Isolation

Insertion of calls to SFI checking function

- Insert BFI instruction at indirect transfer instruction of whole program （code）
- Insert calls to the address-range check at the remaining sites.（data）

```
d0: ldr    r0, [sp+8]
d4: str    r4, [r0, #12]
d8: str    r3, [r0, #8]
dc: str    r2, [r0, #4]
e0: str    r1, [r0]
e4: push   {lr}
e8: mov    r8, r0
ec: bl     str_check
f0: pop    {lr}
f4: mov    r8, 0
f8: bfi    r8,lr,#0,24
fc: bx     r8
```

SFI check site

Insert **address-range** check after check site

Insert BFI before indirect Transfer

Indirect transfer

Outline

1 Background and motivation

2 Design

3 Evaluation

Evaluation

Evaluation setup

- Platform: Raspberry Pi 3B prover; x86 workstation compiler/verifier
- Targets: ArduCopter, ArduRover, PX4, and four embedded apps

		Target Operation	Function count
3 Target Programs	ArduCopter	Update_xy_controller	179
	ArduRover	AC_P_2D::update_all	30
	PX4	_velocityControl	69
4 Embedded apps (from OAT)	Syringe Pump (SP)		
	House Alarm (HA)		
	Light Controller (LC)		
	Remote Movement Controller (RM)		

Overhead

- Average runtime overhead: **24.3%** with BLAKE2s
- On all autopilot systems, ARTO satisfy the real-time constraints
- On autopilots, ARTO reduces execution time by 17.6%-94.2% vs. OAT/ARI
- ARTO reduces runtime overhead by 35.3%-86.1% vs. OP-DFI on Embench test suite

Target Program	Prover-Side Execution Time				
		xxHash		BLAKE2s	
	Orig.(μs)	+ARTO(μs)	Overhead	+ARTO(μs)	Overhead
ArduCopter	522	741	42.0%	815	56.1%
ArduRover	1,106	1,239	12.0%	1,276	15.4%
PX4	340	428	25.9%	506	48.8%
SP	173,305	174,821	0.9%	174,997	1.0%
HA	1,394	1,659	19.0%	1,672	19.9%
LC	5,379	6,841	27.2%	6,927	28.8%
RM	10,019,911	10,022,248	0.02%	10,026,700	0.1%

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

Overhead

- Average runtime overhead: 24.3% with BLAKE2s
- On all autopilot systems, ARTO satisfy the real-time constraints
- On autopilots, ARTO reduces execution time by 17.6%-94.2% vs. OAT/ARI
- ARTO reduces runtime overhead by 35.3%-86.1% vs. OP-DFI on Embench test suite

Target Program	Prover-Side Execution Time				
		xxHash		BLAKE2s	
	Orig.(μs)	+ARTO(μs)	Overhead	+ARTO(μs)	Overhead
ArduCopter	522	741	42.0%	815	56.1%
ArduRover	1,106	1,239	12.0%	1,276	15.4%
PX4	340	428	25.9%	506	48.8%
SP	173,305	174,821	0.9%	174,997	1.0%
HA	1,394	1,659	19.0%	1,672	19.9%
LC	5,379	6,841	27.2%	6,927	28.8%
RM	10,019,911	10,022,248	0.02%	10,026,700	0.1%

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

Overhead

- Average runtime overhead: 24.3% with BLAKE2s
- On all autopilot systems, ARTO satisfy the real-time constraints
- On autopilots, ARTO reduces execution time by 17.6%-94.2% vs. OAT/ARI
- ARTO reduces runtime overhead by 35.3%-86.1% vs. OP-DFI on Embench test suite

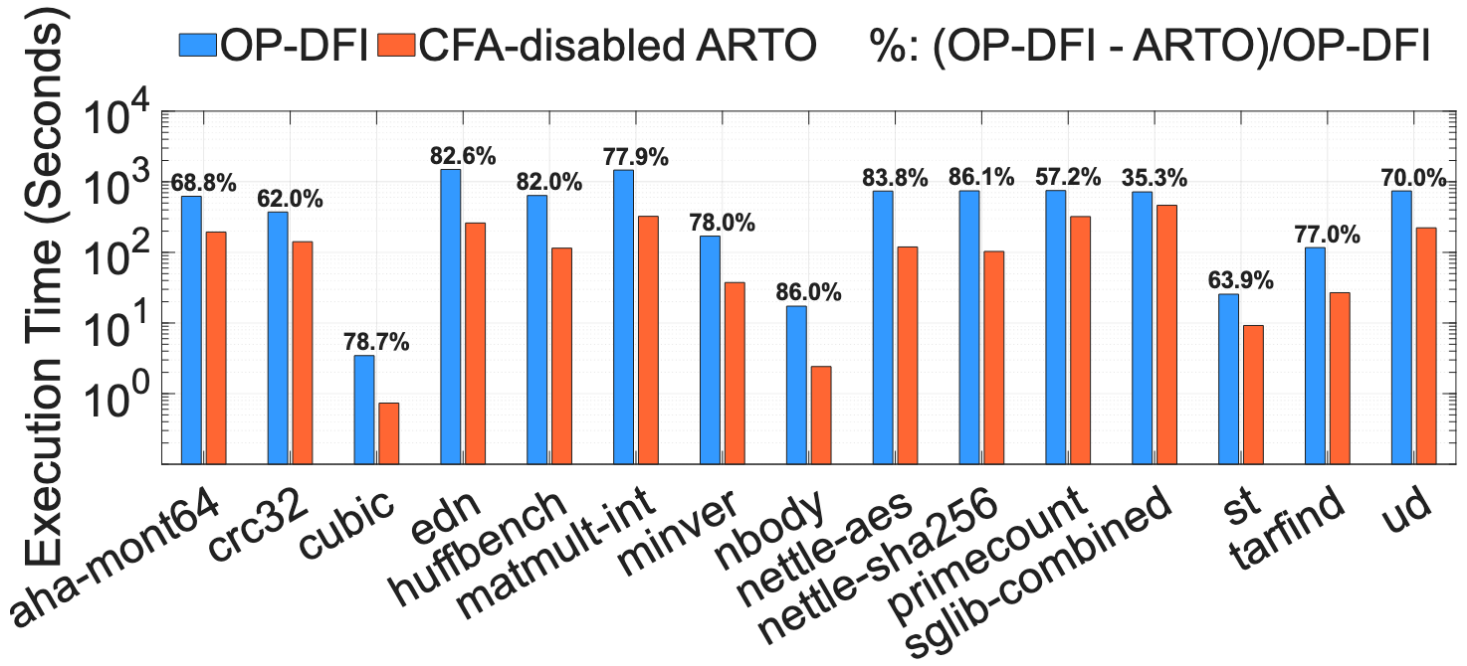
Target Program	Prover-Side Execution Time				
		xxHash		BLAKE2s	
	Orig.(μs)	+ARTO(μs)	Overhead	+ARTO(μs)	Overhead
ArduCopter	522	741	42.0%	815	56.1%
ArduRover	1,106	1,239	12.0%	1,276	15.4%
PX4	340	428	25.9%	506	48.8%
SP	173,305	174,821	0.9%	174,997	1.0%
HA	1,394	1,659	19.0%	1,672	19.9%
LC	5,379	6,841	27.2%	6,927	28.8%
RM	10,019,911	10,022,248	0.02%	10,026,700	0.1%

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

Overhead

- Average runtime overhead: 24.3% with BLAKE2s
- On all autopilot systems, ARTO satisfy the real-time constraints
- On autopilots, ARTO reduces execution time by 17.6%-94.2% vs. OAT/ARI
- ARTO reduces runtime overhead by 35.3%-86.1% vs. OP-DFI on Embench test suite



Attack Mitigation Effectiveness

PoC attacks on 4 targets

Attack	ArduCopter	aha-mont64	Robotic Arm's control loop	PX4
Control-flow hijacking (CFI/CFA)				
① CFG-invalid path	●			
② Escape Critical Operation	●			
③ CFG-valid path (wrong context)	●			
④ CFG-invalid path		●		
⑤ CFG-invalid path using CVE-2025-15150				●
Data-only attacks (DFI)				
⑥ Read-target address manipulation			●	
⑦ Direct value corruption	●	●	●	
⑧ Out-of-bounds data corruption using CVE-2025-5640				●

Attack Mitigation Effectiveness

Attack	ArduCopter	aha-mont64	Robotic Arm's control loop	PX4
Control-flow hijacking (CFI/CFA)				
① CFG-invalid path	●			
② Escape Critical Operation	●			
③ CFG-valid path (wrong context)	●			
④ CFG-invalid path		●		
⑤ CFG-invalid path using CVE-2025-15150				●
Data-only attacks (DFI)				
⑥ Read-target address manipulation			●	
⑦ Direct value corruption	●	●	●	
⑧ Out-of-bounds data corruption using CVE-2025-5640				●

ARTO Successfully detects all attacks

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 μs	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 μs	Yes	0.91 s

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 μs
	ARTO	Yes	< 1 μs
ArduCopter (Attack ⑦)	OAT	Yes	< 1 μs
	ARTO	Yes	< 1 μs
Robotic Arm's Control Loop (Attack ⑥)	OP-DFI	Yes	< 1 μs
	OAT	No	N/A
	ARTO	Yes	< 1 μs
Robotic Arm's Control Loop (Attack ⑦)	OP-DFI	No	N/A
	OAT	Yes	< 1 μs
	ARTO	Yes	< 1 μs
PX4 (Attack ⑧)	OP-DFI	Yes	< 1 μs
	ARTO	Yes	< 1 μs

Attack Mitigation Effectiveness

Attack	ArduCopter	aha-mont64	Robotic Arm's control loop	PX4
Control-flow hijacking (CFI/CFA)				
① CFG-invalid path	●			
② Escape Critical Operation	●			
③ CFG-valid path (wrong context)	●			
④ CFG-invalid path		●		
⑤ CFG-invalid path using CVE-2025-15150				●
Data-only attacks (DFI)				
⑥ Read-target address manipulation			●	
⑦ Direct value corruption	●	●	●	
⑧ Out-of-bounds data corruption using CVE-2025-5640				●

Local detection is immediate for undefined-path attacks and fast for fail-safe triggering

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

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	$< 1\ \mu s$

Attack Mitigation Effectiveness

Attack	ArduCopter	aha-mont64	Robotic Arm's control loop	PX4
Control-flow hijacking (CFI/CFA)				
① CFG-invalid path	●			
② Escape Critical Operation	●			
③ CFG-valid path (wrong context)	●			
④ CFG-invalid path		●		
⑤ CFG-invalid path using CVE-2025-15150				●
Data-only attacks (DFI)				
⑥ Read-target address manipulation			●	
⑦ Direct value corruption	●	●	●	
⑧ Out-of-bounds data corruption using CVE-2025-5640				●

ARTO detects both direct value corruption and read-target manipulation; data-only detections take less than 1 μs.

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

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	$< 1\ \mu s$

Real-Time Impact

- WCET analysis uses EVT (Extreme Value Theory) and MAST schedulability analysis
- Both xxHash- and BLAKE2s-based ARTO **remain schedulable**

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

THANKS

Available:

https://github.com/zz-fz-john/ARTO-Project

https://doi.org/10.5281/zenodo.20355901

Thanks for listening