SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems (Accepted at ACM CCS 2023)

Xi Tan and Ziming Zhao
CactiLab, Department of Computer Science and Engineering, University at Buffalo
{xitan, zimingzh}@buffalo.edu

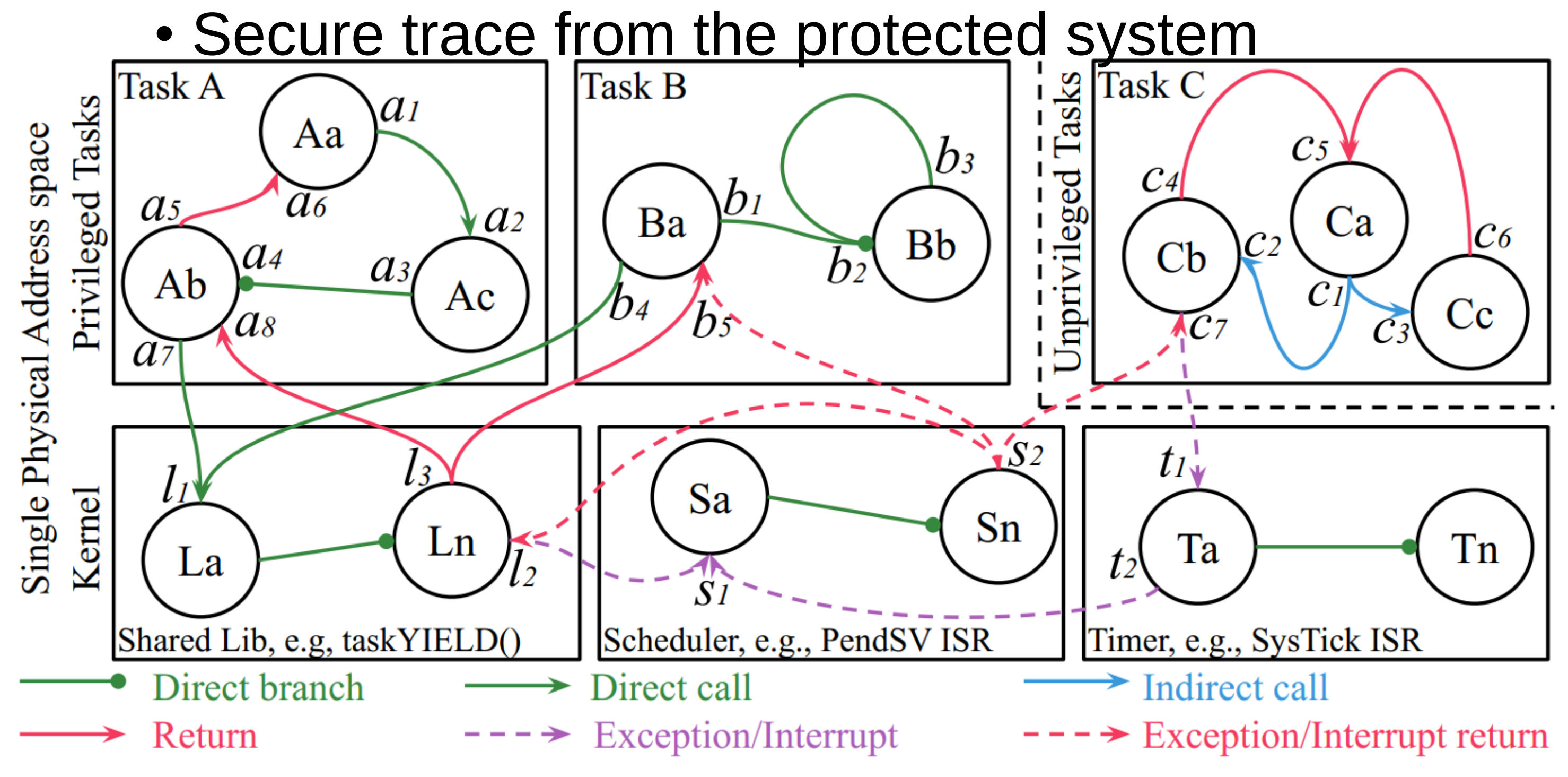
① Introduction

Microcontroller-based embedded systems are often programmed in low-level languages and are vulnerable to control-flow hijacking. But inlined control-flow integrity (CFI) enforcement solutions increase the binary size and change the memory layout. Trace-based control-flow violation detection (CFVD) offers an alternative, but existing solutions are **application-oriented**, requiring kernel modifications to store and analyze the trace, limiting their use to monitor privileged codes.

② System-oriented CFVD

Monitor control-flow transfers both within and among privileged and unprivileged components.

- **Interrupt-aware**
  - Interrupts and exceptions occur asynchronously and cannot be anticipated through static analysis or dynamic training. E.g., <c_7, t_1>
- **Scheduling-aware**
  - Scheduler may resume any running tasks' execution. E.g., both $\langle s_2, b_5\rangle$ and $\langle s_2, c_7\rangle$ are legitimate control-flow transfers.
- **Secure hardware tracing**
  - Prevent privileged but potentially compromised system to disable tracing
- **Secure trace storage and analysis**
  - Secure trace from the protected system



SCFVD verifies each indirect control-flow transfer must match an edge in the interprocedural CFG (G_S) or the destination must match an address in the interrupt ISR address list (I_K) or the set of task entry or re-entry list (Y_T)

System-oriented CFVD (SCFVD). Given the trace $R_{\mathcal{S}} = (r_0, r_1, \ldots, r_n)$ of a system $\mathcal{S}$ including a kernel $\mathcal{K}$ and tasks $\mathcal{T}$, SCFVD verifies that $r_i \in E_{\mathcal{S}} \vee r_i.d \in I_{\mathcal{K}} \bigcup Y_{\mathcal{T}}, \forall i \in \{0, 1, \ldots, n\}$.

③ System and Threat Model

- The system features a hardware trace unit. Filtering capabilities are not required.
- The system supports TEE and secure boot for code integrity.
- Attackers can exploit privileged code bugs of the protected system via memory corruptions.

④ Design Overview

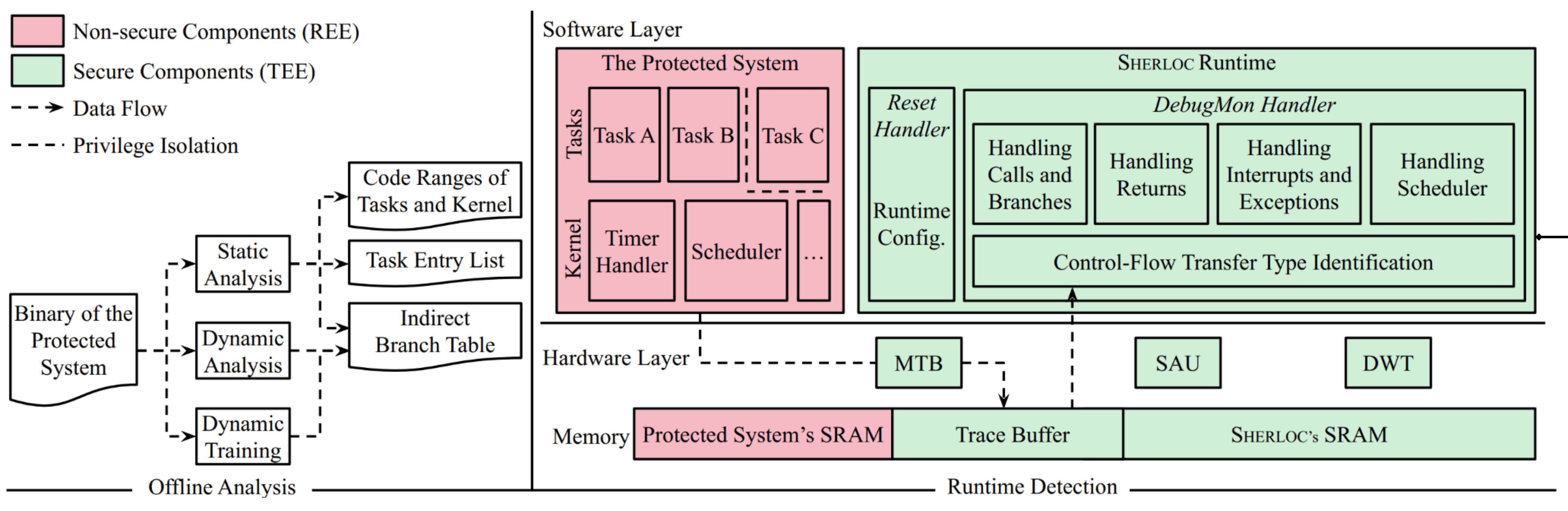


Figure 2: SHERLOC comprises offline analysis and runtime configuration and enforcement modules. The unmodified protected system program runs in the non-secure state, whereas SHERLOC runtime modules execute in the secure state.

⑤ Runtime Detection Policy

The approach that SHERLOC takes for handling each type of dereferenced instruction in the trace. $\langle s, d\rangle$: a standard trace record. $(\langle s_1, d_1\rangle, \langle s_2, d_2\rangle)$: a pair of interrupt or exception return trace records. IBT: Indirect branch table. VT: non-secure state vector table. RCS: the current task- or kernel-specific reconstructed call stack. Y_T: task entry and re-entry address list.

	Type	Instruction(s)	Ins. Size	How to Identify the Type?	SHERLOC Actions
Bare-metal System and RTOS Cases	Direct branch (§4.4.1)	`B{cond} #imm`	2/4	The dereferenced instruction	Skip the record
	Direct call (§4.4.1)	`BL{cond} #imm`	4	The dereferenced instruction	RCS.push(s + 4)

	Indirect branch (§4.4.1)	`BX{cond} Rx`
		`TBB/TBH {PC, ...}`
			2
			4
				The dereferenced instruction	*if* $\langle s, d\rangle \notin$ IBT, reset

	Indirect call (§4.4.1)	`BLX Rx`	2	The dereferenced instruction	*if* $\langle s, d\rangle \notin$ IBT, reset; *else* RCS.push(s + 2)

	Function return (§4.4.2)	`BX LR`
		`POP {..., PC}`
		`LDM SP!, {..., PC}`
			2/4	The dereferenced instruction	*if* $d \neq$ RCS.pop(), reset

	Sync. exception (§4.4.3)	`SVC #imm`	2	s[A-bit]	*if* $d \notin$ VT, reset; *else if* $d \neq$ PendSV, RCS.push(s)
	Non-PendSV async. interrupt (§4.4.3)	N/A	N/A	s[A-bit]	*if* $d \notin$ VT, reset; *else if* $d \neq$ PendSV, RCS.push(s)

	Non-PendSV ISR return (§4.4.4)	`BX LR`
		`POP {..., PC}`
		`LDM SP!, {..., PC}`
			2/4	The dereferenced instruction and (d_1 == EXC_RETURN ∧ s_2 == EXC_RETURN)	*if* bare-metal and $d_2 \neq$ RCS.top(), reset;
					else if bare-metal and d_2 == RCS.top(), RCS.pop();
					else go to PendSV ISR return handling

RTOS-only Cases	PendSV async. interrupt (§4.4.5)	N/A	N/A	s[A-bit]	*if* d == PendSV, $Y_{\mathcal{T}}$.add(s) and $Y_{\mathcal{T}}$.add(RCS.pop())

	PendSV ISR return (§4.4.6)	`BX LR`
		`POP {..., PC}`
		`LDM SP!, {..., PC}`
			2/4	The dereferenced instruction and (d_1 == EXC_RETURN ∧ s_2 == EXC_RETURN)	*if* $d_2 \notin Y_{\mathcal{T}}$, reset;
					if d_2 is in a shared library,
					and assuming the next trace record is $\langle s_n, d_n\rangle$,
					and $d_n \notin Y_{\mathcal{T}}$, reset

⑥ Running Example on FreeRTOS

[] represents RCS with the top on the right-hand side. Black [] represents the active RCS, and gray [] represents an inactive RCS.

Trace Buffer	Runtime Enforcement	RCS for Task A	RCS for Task B	Y_T
$\langle a_3, a_4\rangle$	Direct branch: skip	$[a_1+4]$	[...]	$\{b_4+4, l_2, \ldots\}$
$\langle a_1, a_2\rangle$	Direct call: RCS.push($\boldsymbol{a_1}$ + 4)	$[\boldsymbol{a_1+4}]$	[...]	$\{b_4+4, l_2, \ldots\}$
$\langle a_5, a_6\rangle$	Function return: a_6 == RCS.pop()	[], ~~a_1+4~~	[...]	$\{b_4+4, l_2, \ldots\}$
$\langle l_3, a_8\rangle$	Function return: a_8 *is in* Y_T ($\boldsymbol{a_8} == \boldsymbol{a_7}+4$)	$[a_1+4]$	[...]	$\{b_4+4, l_2, \ldots\}$, ~~a_7+4~~
$\langle$EXC_RETURN, $l_2\rangle$ $\langle s_2$, EXC_RETURN$\rangle$	PendSV ISR return: l_2 *is in* Y_T	$[a_1+4]$	[...]	$\{b_4+4, l_2, a_7+4, \ldots\}$, ~~$l_2$~~
$\langle l_2, s_1\rangle$	s_1 is PendSV ISR address: Y_T.add($\boldsymbol{l_2}$) and Y_T.add($\boldsymbol{b_4}$ + 4)	$[a_1+4]$	[...], ~~b_4+4~~	$\{\boldsymbol{l_2}, \boldsymbol{b_4+4}, l_2, a_7+4, \ldots\}$
$\langle b_4, l_1\rangle$	Direct call: RCS.push($\boldsymbol{b_4}$ + 4)	$[a_1+4]$	[..., $\boldsymbol{b_4+4}$]	$\{l_2, a_7+4, \ldots\}$
$\langle$EXC_RETURN, $b_5\rangle$ $\langle s_2$, EXC_RETURN$\rangle$	PendSV ISR return: b_5 *is in* Y_T	$[a_1+4]$	[...]	$\{l_2, a_7+4, \ldots\}$, ~~b_5~~
…	…	$[a_1+4]$	[...]	$\{b_5, l_2, a_7+4, \ldots\}$
$\langle l_2, s_1\rangle$	s_1 is PendSV ISR address: Y_T.add($\boldsymbol{l_2}$) and Y_T.add($\boldsymbol{a_7}$ + 4)	$[a_1+4]$, ~~a_7+4~~	[...]	$\{b_5, \boldsymbol{l_2}, \boldsymbol{a_7+4}, \ldots\}$
$\langle a_7, l_1\rangle$	Direct call: RCS.push($\boldsymbol{a_7}$ + 4)	$[a_1+4, \boldsymbol{a_7+4}]$	[...]	$\{b_5, \ldots\}$
$\langle a_3, a_4\rangle$	Direct branch: skip	$[a_1+4]$	[...]	$\{b_5, \ldots\}$
$\langle a_1, a_2\rangle$	Direct call: RCS.push($\boldsymbol{a_1}$ + 4)	$[\boldsymbol{a_1+4}]$	[...]	$\{b_5, \ldots\}$
…	…	[]	[...]	$\{b_5, \ldots\}$

🟩 Direct transfer 🟥 Function return 🟪 ISR entry ▨ ISR return

University at Buffalo The State University of New York

CactiLab Code Repo

This material is based upon work supported in part by National Science Foundation (NSF) grants (2237238 and 2037798) and a National Centers of Academic Excellence in Cybersecurity grant (H98230-22-1-0307).