Alligator in Vest: A Practical Failure-Diagnosis Framework via Arm Hardware Features

Yiming Zhang[1,2], Yuxin Hu[1], Haonan Li[1] , Wenxuan Shi[1] , Zhenyu Ning[3,1], Xiapu Luo[2], Fengwei Zhang[1]

[1]Southern University of Science and Technology, [2]The Hong Kong Polytechnic University, [3]Hunan University

Failure Diagnosis in Production

- Software failures are unavoidable in production environments

Failure Diagnosis in Production

- Software failures are unavoidable in production environments
- Existing failure diagnosis approaches (i.e., Postmortem Analysis and Record&Replay based approaches) usually is unsatisfied in production:

Failure Diagnosis in Production

- Software failures are unavoidable in production environments
- Existing failure diagnosis approaches (i.e., Postmortem Analysis and Record&Replay based approaches) usually is unsatisfied in production:
  - ✕ The complexity and limited information impede analysis using memory crashed coredump.

Failure Diagnosis in Production

- Software failures are unavoidable in production environments
- Existing failure diagnosis approaches (i.e., Postmortem Analysis and Record&Replay based approaches) usually is unsatisfied in production:
    - ✗ The complexity and limited information impede analysis using memory crashed coredump.
    - ✗ Record&Replay incurs heavy overhead.

Failure Diagnosis in Production

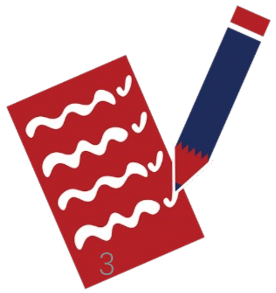
- Software failures are unavoidable in production environments
- Existing failure diagnosis approaches (i.e., Postmortem Analysis and Record&Replay based approaches) usually is unsatisfied in production:
  - ✗ The complexity and limited information impede analysis using memory crashed coredump.
  - ✗ Record&Replay incurs heavy overhead.
  - ⚠ Broadly studied on x86 platforms [1,2,5,6], but still is an open problem on Arm architecture

Failure Diagnosis in Production

- Software failures are unavoidable in production environments
- Existing failure diagnosis approaches (i.e., Postmortem Analysis and Record&Replay based approaches) usually is unsatisfied in production:
  - ✗ The complexity and limited information impede analysis using memory crashed coredump.
  - ✗ Record&Replay incurs heavy overhead.
  - ⚠ Broadly studied on x86 platforms [1,2,5,6], but still is an open problem on Arm architecture

➢ To overcome the limitations, we propose a novel hardware-assisted framework on Arm named ***Investigator*** for failure diagnosis in production

Agenda



Agenda

- Background
- Investigator: Analyzer for Failure Diagnosis in Production Environments
- Experimental Results
- Summary and Future Work

Background

- ETM (Embedded Trace Microcell)
  - It is a hardware feature of Arm.

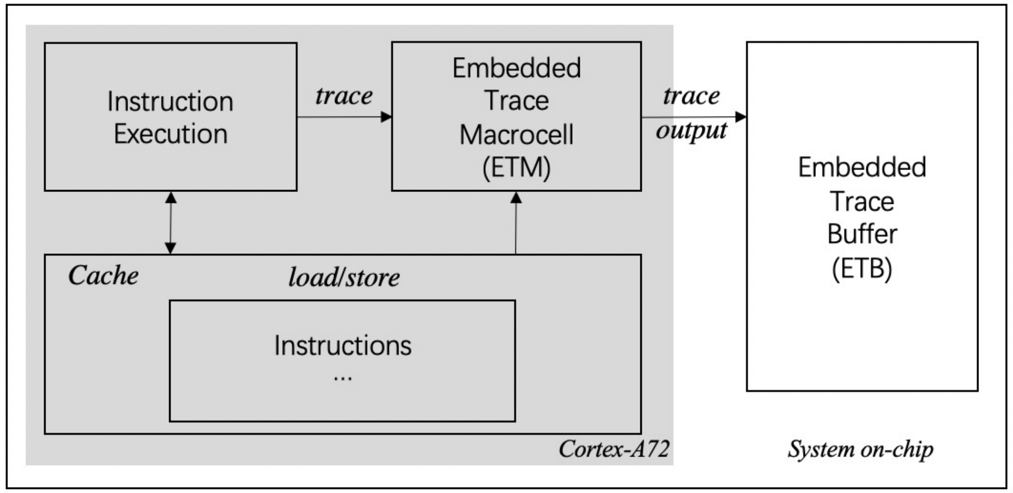


Figure. ETM.

Background

- ETM (Embedded Trace Microcell)
  - It is a hardware feature of Arm.
  - It traces the instructions executed by CPU with almost no overhead.

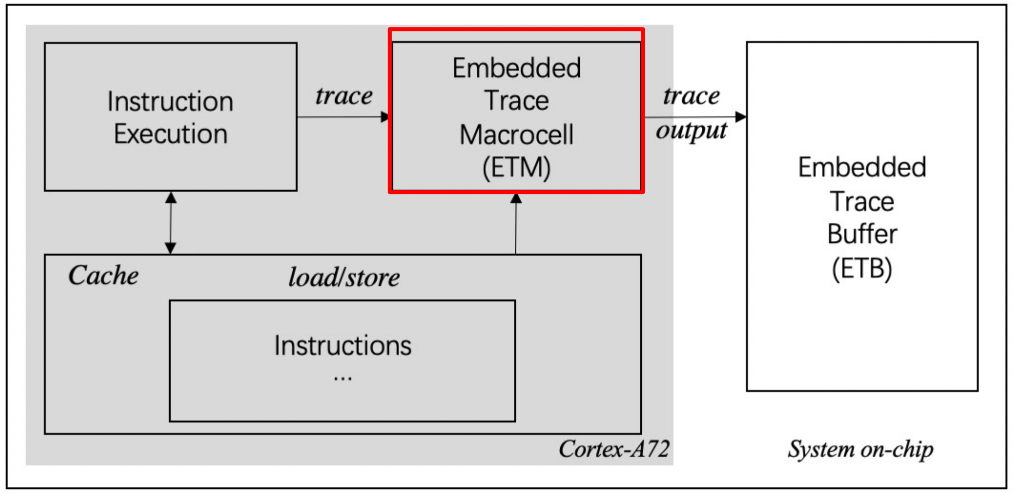


Figure. ETM.

Background

- ETM (Embedded Trace Microcell)
  - It is a hardware feature of Arm.
  - It traces the instructions executed by CPU with almost no overhead.

- PMU (Performance Monitor Unit)
  - It is a group of counters that can count any of the events available in the core.

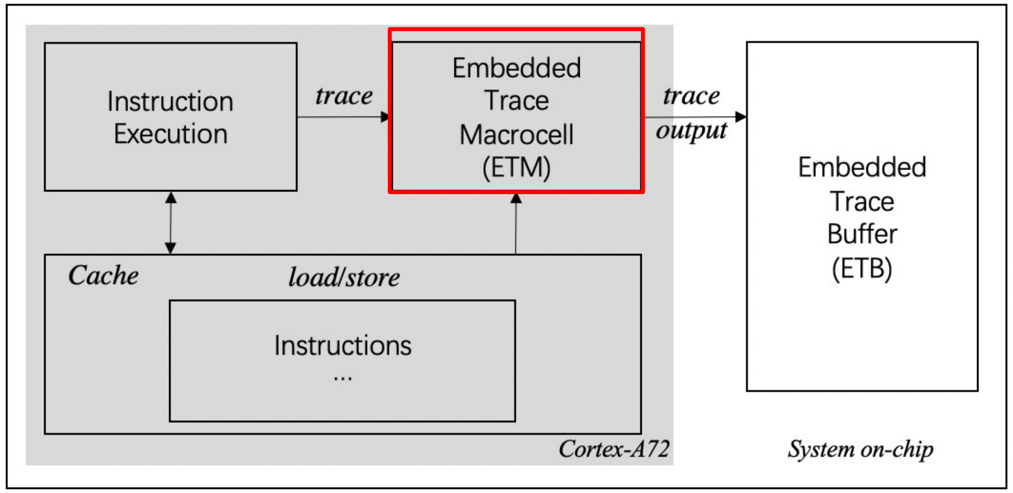


Figure. ETM.

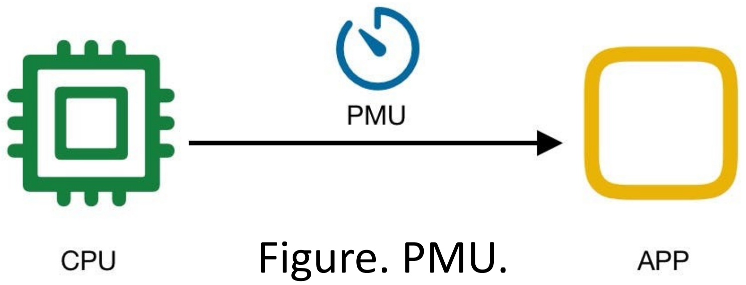


Figure. PMU.

Background

- ETM (Embedded Trace Microcell)
  - It is a hardware feature of Arm.
  - It traces the instructions executed by CPU with almost no overhead.

- PMU (Performance Monitor Unit)
  - It is a group of counters that can count any of the events available in the core.
  - PMU interrupt is generated when the PMU counter overflows.

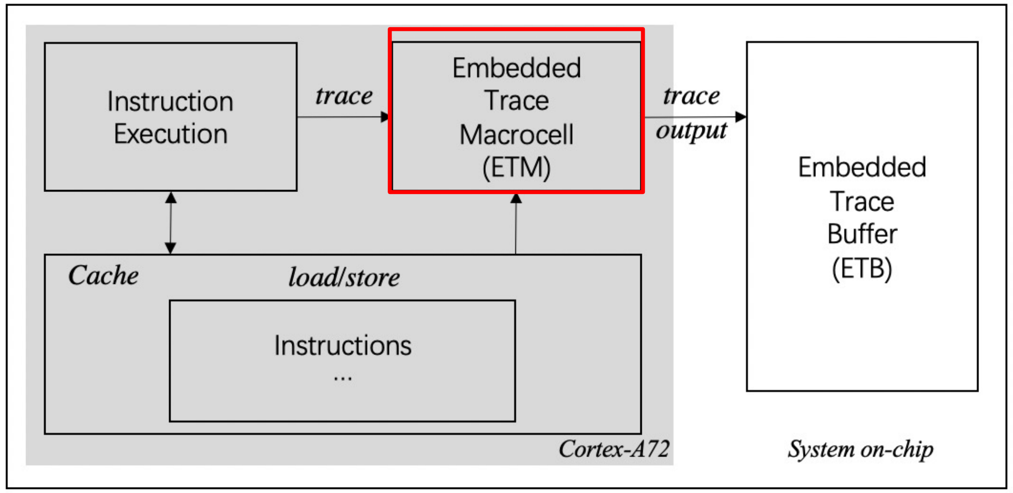


Figure. ETM.

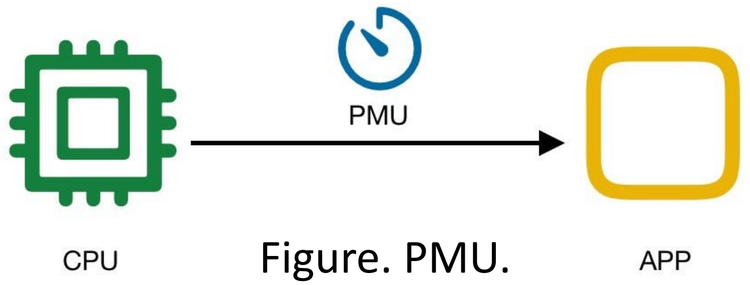


Figure. PMU.

Investigator Architecture

Investigator Architecture

- It consists of two modules:

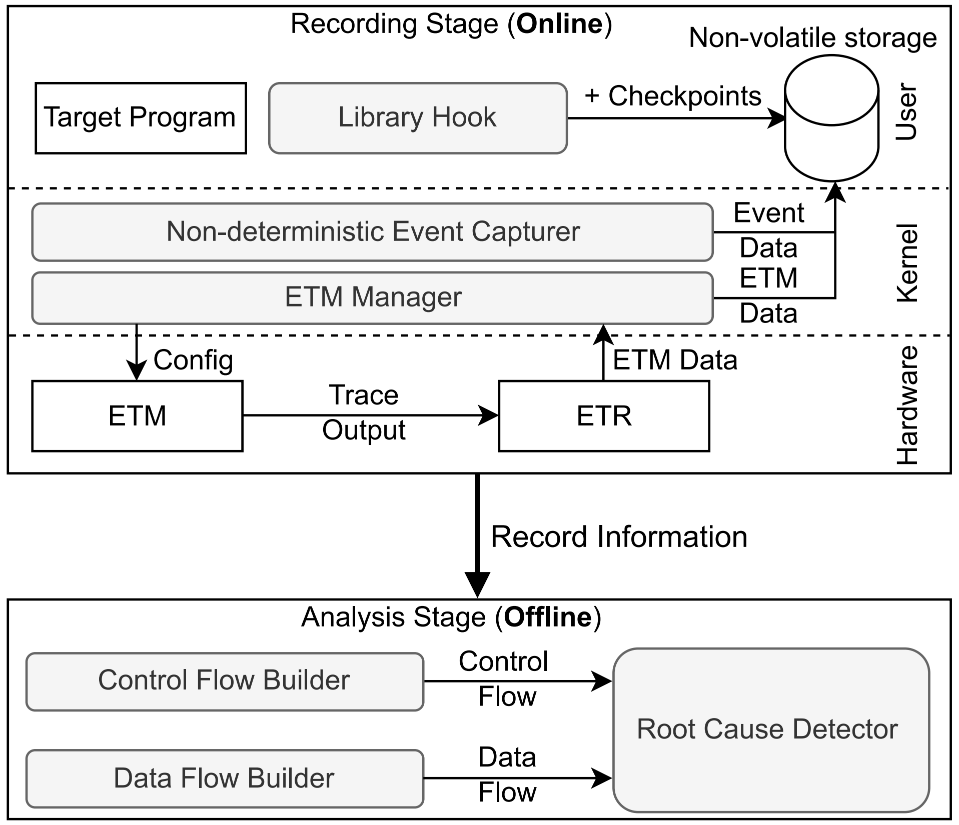


Figure. Design of Investigator

Investigator Architecture

- It consists of two modules:
  - An record module with software-based collection in production host.

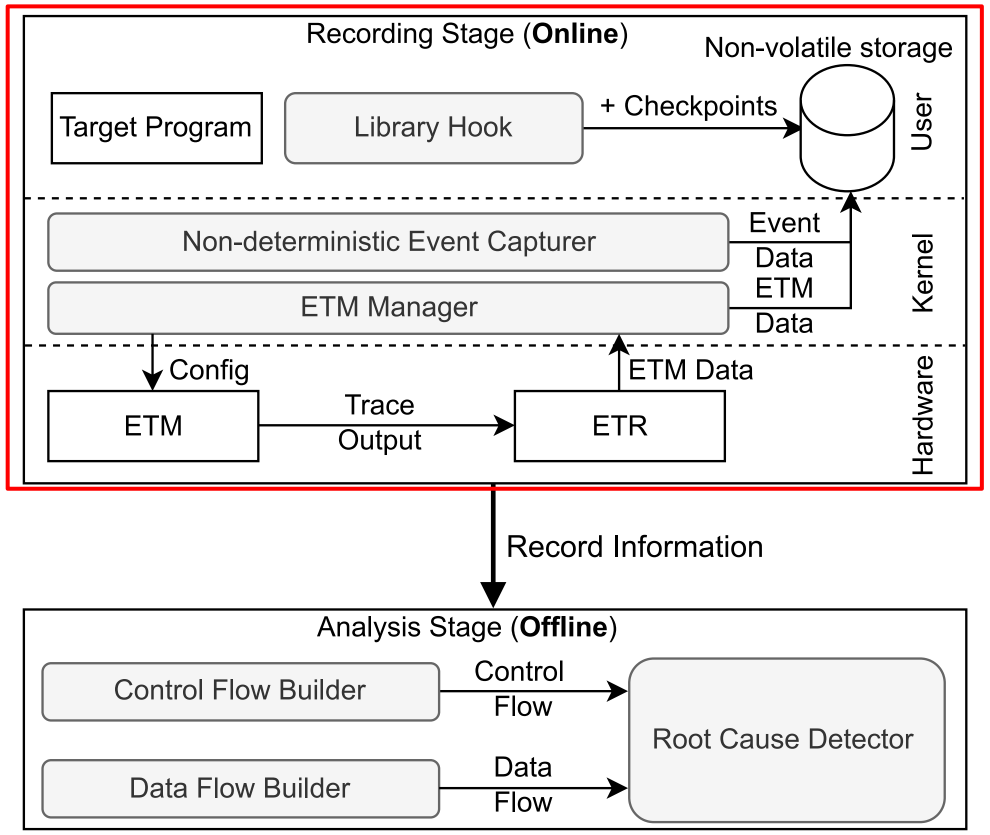


Figure. Design of Investigator

Investigator Architecture

- It consists of two modules:
  - An record module with software-based collection in production host.
  - An analysis module implementing execution recovery and analysis methods in offline server.

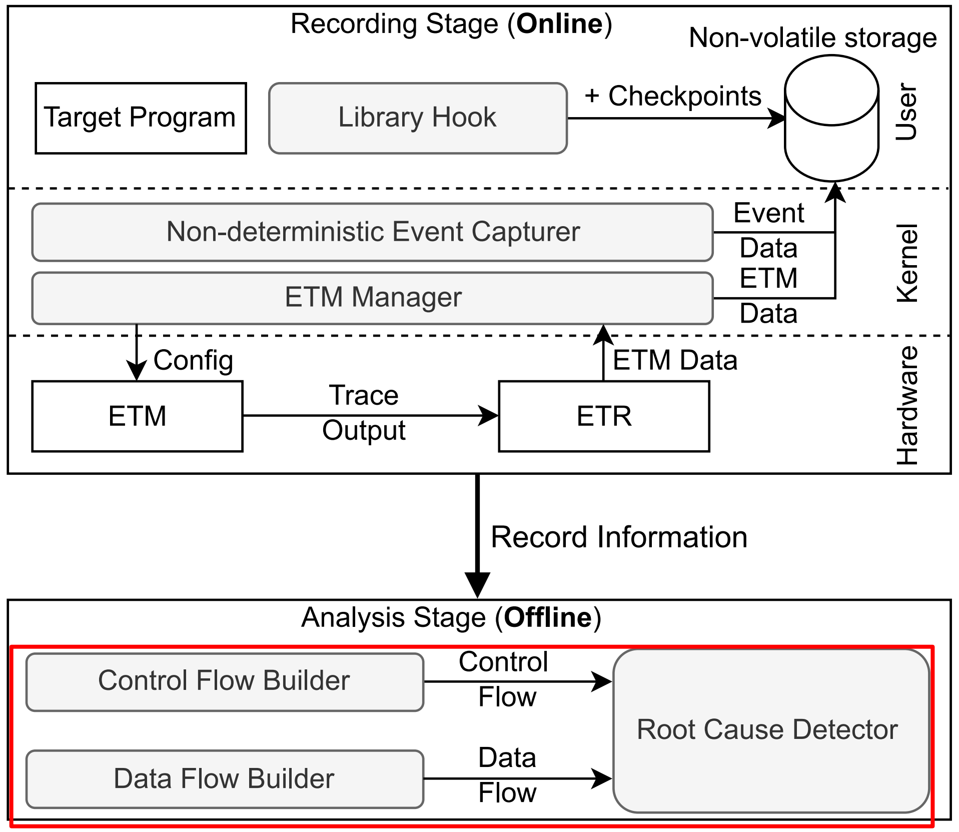


Figure. Design of Investigator

Record Module: ETM Manager

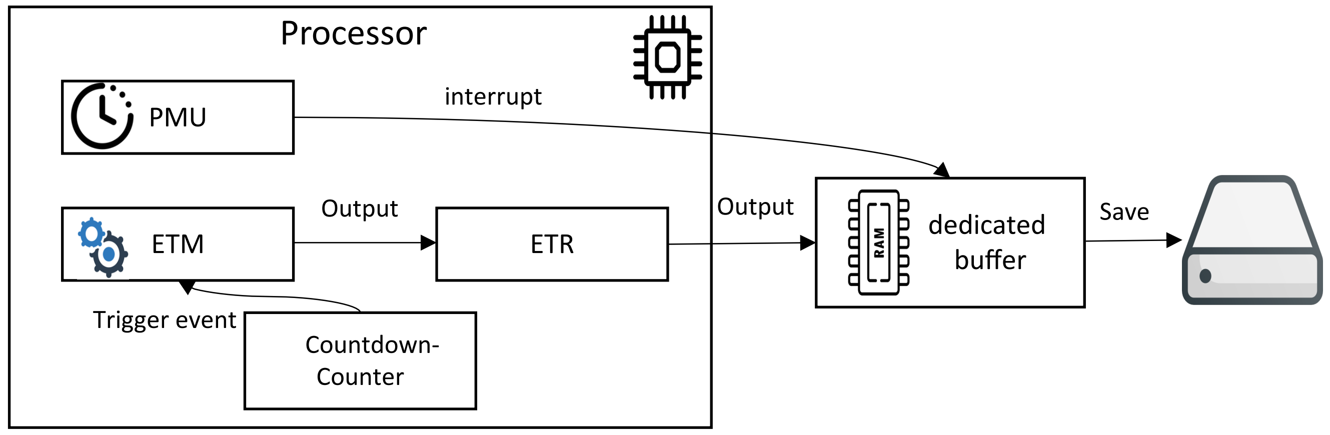


Figure. ETM Manager

Record Module: ETM Manager

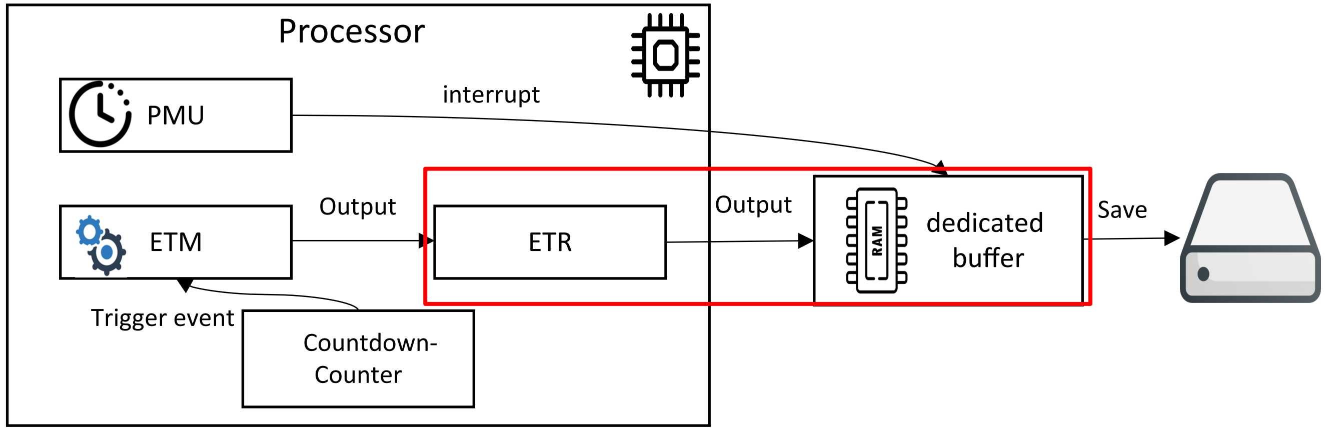


Figure. ETM Manager

Record Module: ETM Manager

- ➢ Accurate trace without losing data

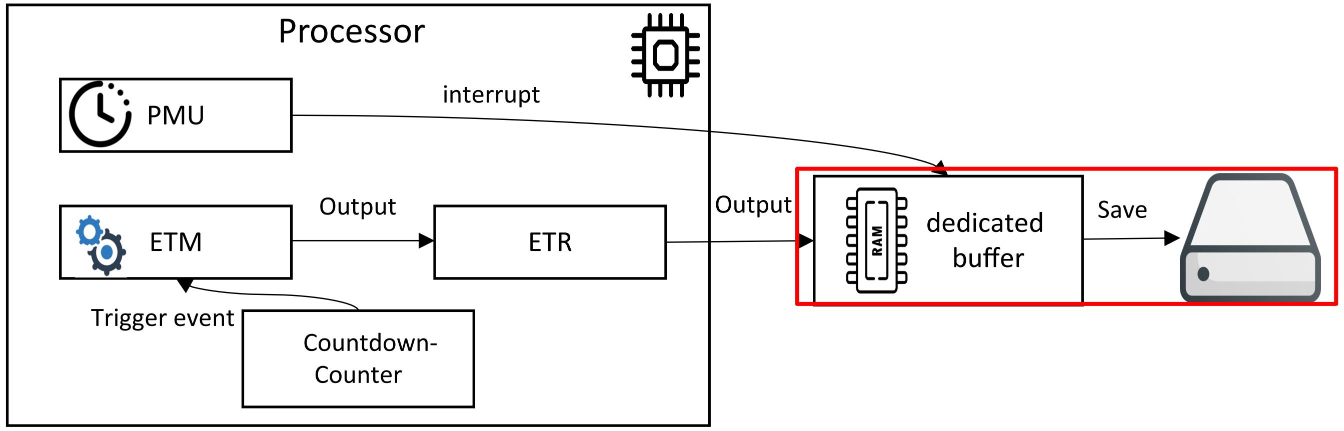


Figure. ETM Manager

Record Module: ETM Manager

- ➢ Accurate trace without losing data
  - Using PMU to count instruction number to estimate the size of ETM buffer

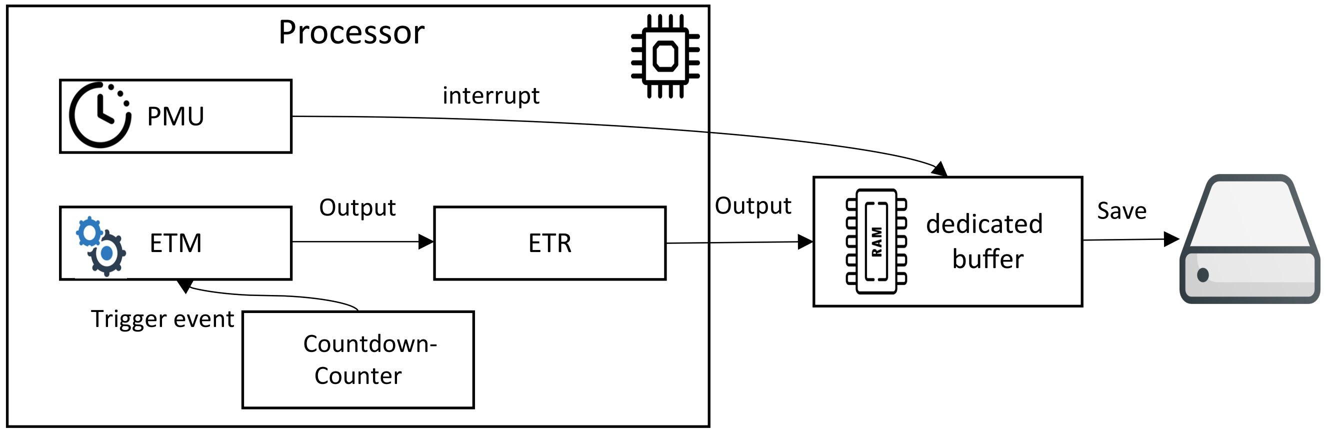


Figure. ETM Manager

Record Module: ETM Manager

- Accurate trace without losing data
  - Using PMU to count instruction number to estimate the size of ETM buffer
  - Trace dump in PMI handler

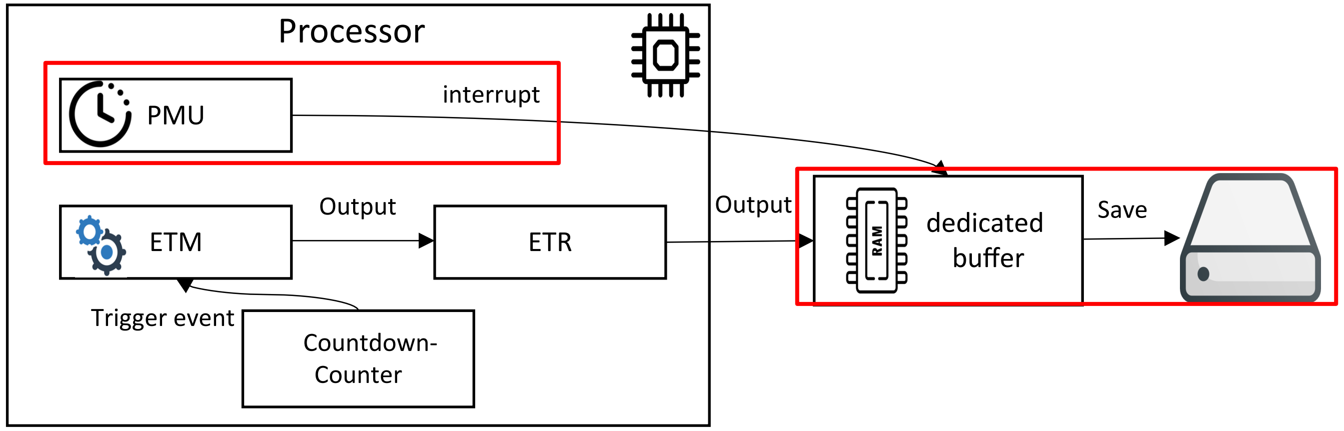


Figure. ETM Manager

Record Module: ETM Manager

- ➢ Accurate trace without losing data
  - Using PMU to count instruction number to estimate the size of ETM buffer
  - Trace dump in PMI handler
- ➢ Fine-grained timestamps identifying the order across threads

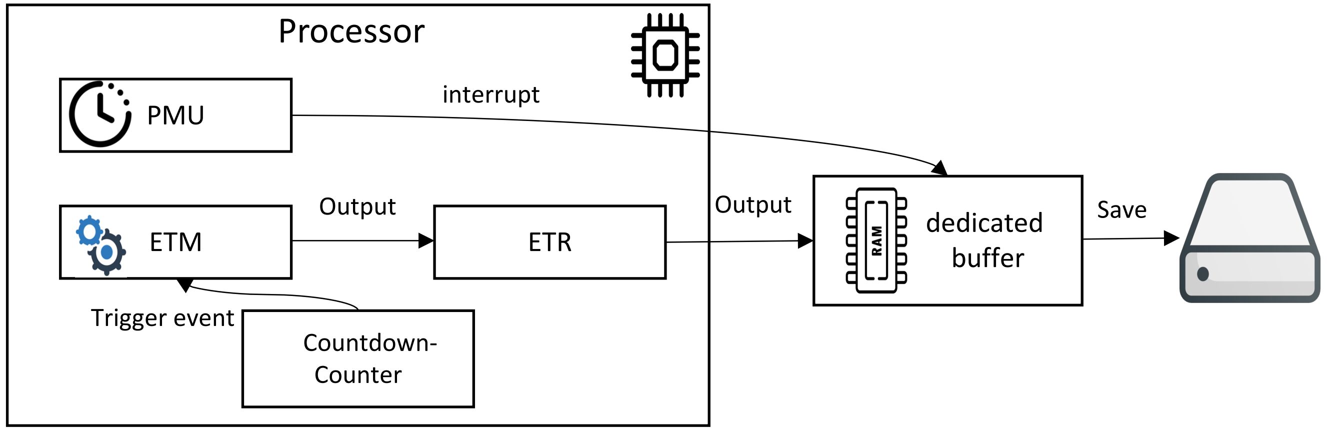


Figure. ETM Manager

Record Module: ETM Manager

- ➢ Accurate trace without losing data
  - Using PMU to count instruction number to estimate the size of ETM buffer
  - Trace dump in PMI handler
- ➢ Fine-grained timestamps identifying the order across threads
  - Countdown-Counter as an external source to maximize ETM timestamp generation.

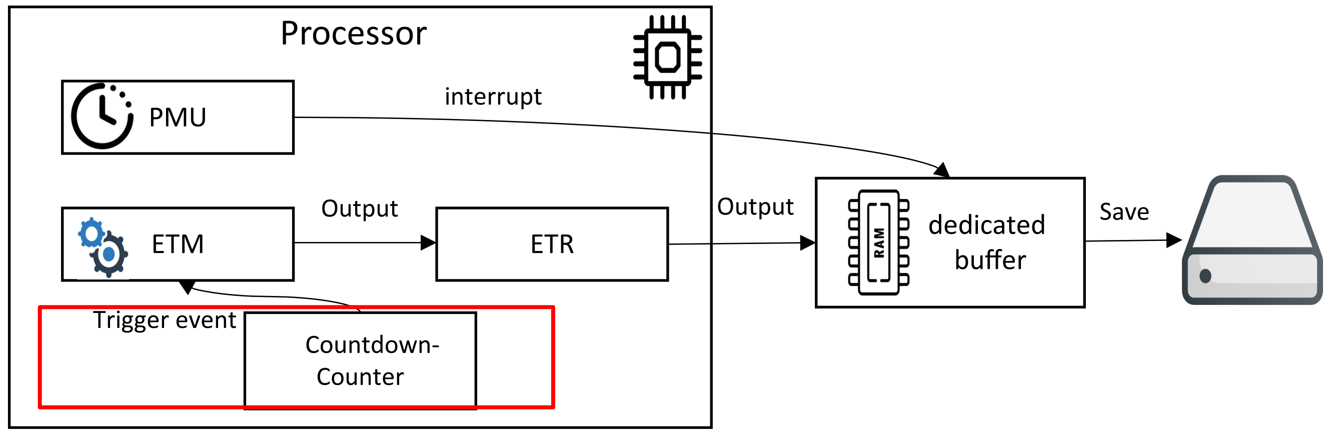


Figure. ETM Manager

Record Module: ETM Manager

- ➢ Accurate trace without losing data
  - Using PMU to count instruction number to estimate the size of ETM buffer
  - Trace dump in PMI handler
- ➢ Fine-grained timestamps identifying the order across threads
  - Countdown-Counter as an external source to maximize ETM timestamp generation.
- ➢ Filtered tracing
  - Context ID and Address range.

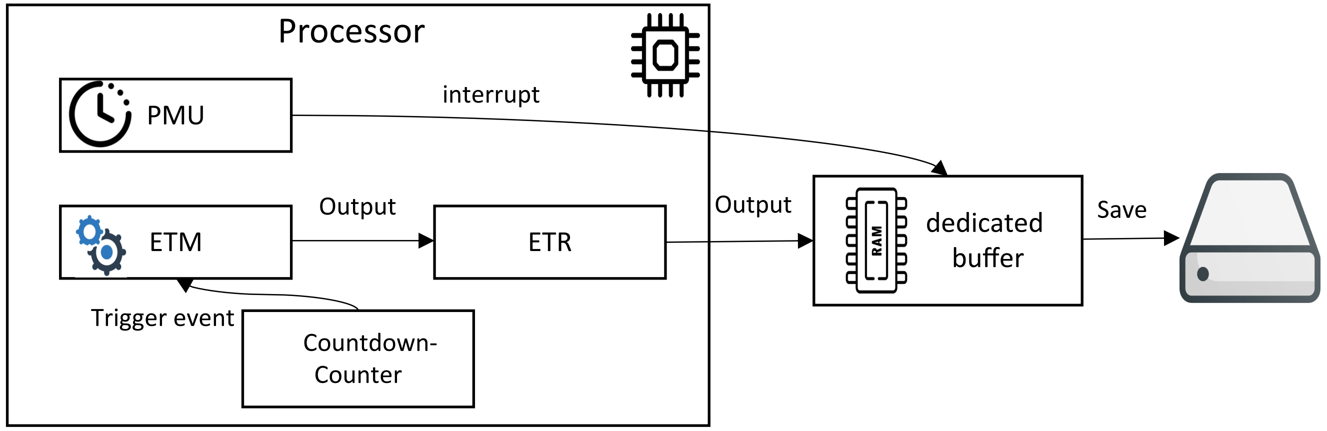


Figure. ETM Manager

Record Module: handle non-deterministic events from syscalls

Trade-off: Provide accurate data flow including syscall impact for failure diagnosis without high overhead

Record Module: handle non-deterministic events from syscalls

Trade-off: Provide accurate data flow including syscall impact for failure diagnosis without high overhead

➢Turn off the ETM for kernel space trace

Record Module: handle non-deterministic events from syscalls

Trade-off: Provide accurate data flow including syscall impact for failure diagnosis without high overhead

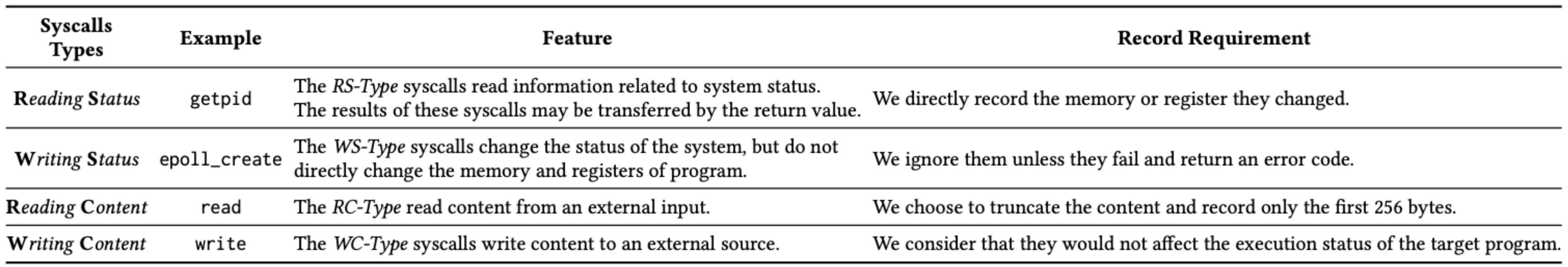
- Turn off the ETM for kernel space trace
- Carputer: Recording the effects of non-deterministic events from syscalls.

Record Module: handle non-deterministic events from syscalls

Trade-off: Provide accurate data flow including syscall impact for failure diagnosis without high overhead

- ➢Turn off the ETM for kernel space trace
- ➢Carputer: Recording the effects of non-deterministic events from syscalls.
    - ➢Record syscalls with different strategies to reduce overhead.

Table. The classification of syscall.



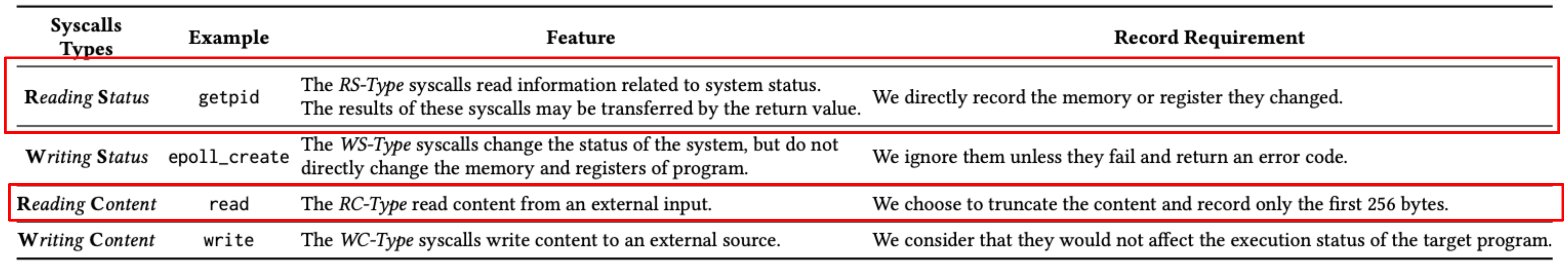
Syscalls Types	Example	Feature	Record Requirement
Reading **S**tatus	`getpid`	The *RS-Type* syscalls read information related to system status. The results of these syscalls may be transferred by the return value.	We directly record the memory or register they changed.
Writing **S**tatus	`epoll_create`	The *WS-Type* syscalls change the status of the system, but do not directly change the memory and registers of program.	We ignore them unless they fail and return an error code.
Reading **C**ontent	`read`	The *RC-Type* read content from an external input.	We choose to truncate the content and record only the first 256 bytes.
Writing **C**ontent	`write`	The *WC-Type* syscalls write content to an external source.	We consider that they would not affect the execution status of the target program.

Record Module: handle non-deterministic events from syscalls

Trade-off: Provide accurate data flow including syscall impact for failure diagnosis without high overhead

- ➢Turn off the ETM for kernel space trace
- ➢Carputer: Recording the effects of non-deterministic events from syscalls.
  - ➢Record syscalls with different strategies to reduce overhead.

Table. The classification of syscall.



Syscalls Types	**Example**	**Feature**	**Record Requirement**
R*eading* **S***tatus*	`getpid`	The *RS-Type* syscalls read information related to system status. The results of these syscalls may be transferred by the return value.	We directly record the memory or register they changed.
W*riting* **S***tatus*	`epoll_create`	The *WS-Type* syscalls change the status of the system, but do not directly change the memory and registers of program.	We ignore them unless they fail and return an error code.
R*eading* **C***ontent*	`read`	The *RC-Type* read content from an external input.	We choose to truncate the content and record only the first 256 bytes.
W*riting* **C***ontent*	`write`	The *WC-Type* syscalls write content to an external source.	We consider that they would not affect the execution status of the target program.

Analysis Module: Execution Flow Recovery

Analysis Module: Execution Flow Recovery

- Control Flow Builder: ETM trace+ binary
    - reconstruct each instruction that the program executes

Analysis Module: Execution Flow Recovery

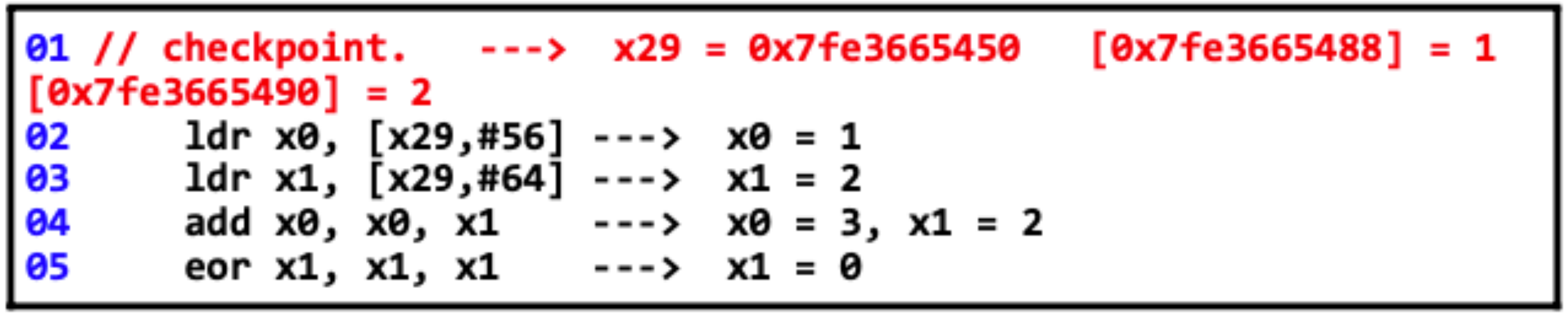
- Control Flow Builder: ETM trace+ binary
    - reconstruct each instruction that the program executes

- Data Flow Builder: reconstruct the data flow

Analysis Module: Execution Flow Recovery

- Control Flow Builder: ETM trace+ binary
  - reconstruct each instruction that the program executes

- Data Flow Builder: reconstruct the data flow
  - infers the state of memory and register after the execution of each instruction based on an initial program state (i.e., checkpoint)



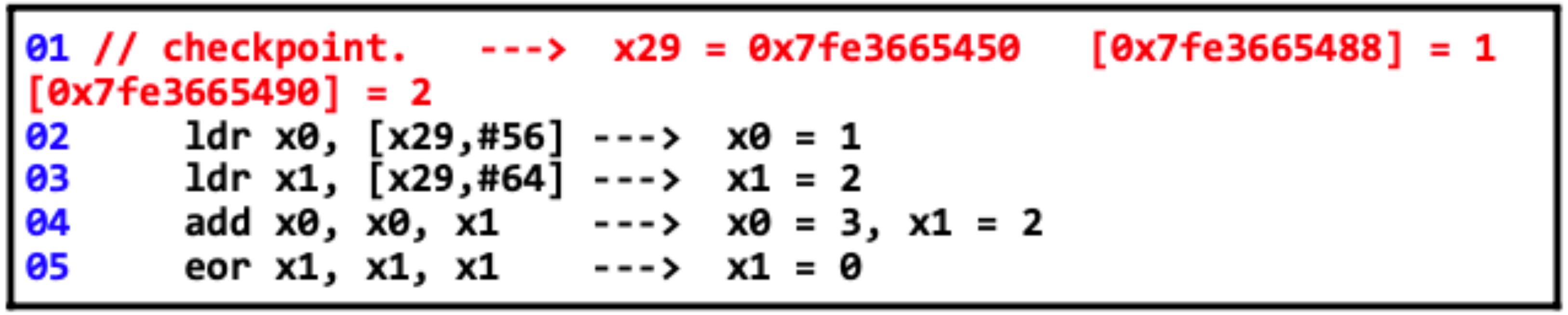
```
01 // checkpoint.   --->  x29 = 0x7fe3665450   [0x7fe3665488] = 1
[0x7fe3665490] = 2
02      ldr x0, [x29,#56] --->  x0 = 1
03      ldr x1, [x29,#64] --->  x1 = 2
04      add x0, x0, x1    --->  x0 = 3, x1 = 2
05      eor x1, x1, x1    --->  x1 = 0
```

Figure. Data flow construction.

Analysis Module: Execution Flow Recovery

- Control Flow Builder: ETM trace+ binary
  - reconstruct each instruction that the program executes

- Data Flow Builder: reconstruct the data flow
  - infers the state of memory and register after the execution of each instruction based on an initial program state (i.e., checkpoint)
  - for syscalls that cannot be inferred, recovers the data flow by parsing recorded information



```
// checkpoint.   --->  x29 = 0x7fe3665450   [0x7fe3665488] = 1
[0x7fe3665490] = 2
     ldr x0, [x29,#56] --->  x0 = 1
     ldr x1, [x29,#64] --->  x1 = 2
     add x0, x0, x1    --->  x0 = 3, x1 = 2
     eor x1, x1, x1    --->  x1 = 0
```

Figure. Data flow construction.

Facilitating Failure Diagnosis Procedure

Facilitating Failure Diagnosis Procedure

- Detector: adapting the existing work [5]
    - Narrow down the cause of a failure from the reconstructed control-data flow.

Facilitating Failure Diagnosis Procedure

- Detector: adapting the existing work [5]
  - Narrow down the cause of a failure from the reconstructed control-data flow.
- E.g., Concurrency failure detection
  - Identify failing address

Facilitating Failure Diagnosis Procedure

- Detector: adapting the existing work [5]
  - Narrow down the cause of a failure from the reconstructed control-data flow.
- E.g., Concurrency failure detection
  - Identify failing address
  - Find alias variables and the memory locations via inclusion-based points-to analysis [4]

Facilitating Failure Diagnosis Procedure

- Detector: adapting the existing work [5]
    - Narrow down the cause of a failure from the reconstructed control-data flow.
- E.g., Concurrency failure detection
    - Identify failing address
    - Find alias variables and the memory locations via inclusion-based points-to analysis [4]
    - Statistical approach using patterns

Atomicity Violation	Order violation
RWR	WR
WWR	RR
RWW	WW
WRW	

Figure. Patterns for concurrency failure prediction.

Facilitating Failure Diagnosis Procedure

- Detector: adapting the existing work [5]
  - Narrow down the cause of a failure from the reconstructed control-data flow.
- E.g., Concurrency failure detection
  - Identify failing address
  - Find alias variables and the memory locations via inclusion-based points-to analysis [4]
  - Statistical approach using patterns
  - Eliminate patterns that present in normal executions without failure

Atomicity Violation	Order violation
RWR	WR
WWR	RR
RWW	WW
WRW	

Figure. Patterns for concurrency failure prediction.

How is the overhead incurred by Investigator?

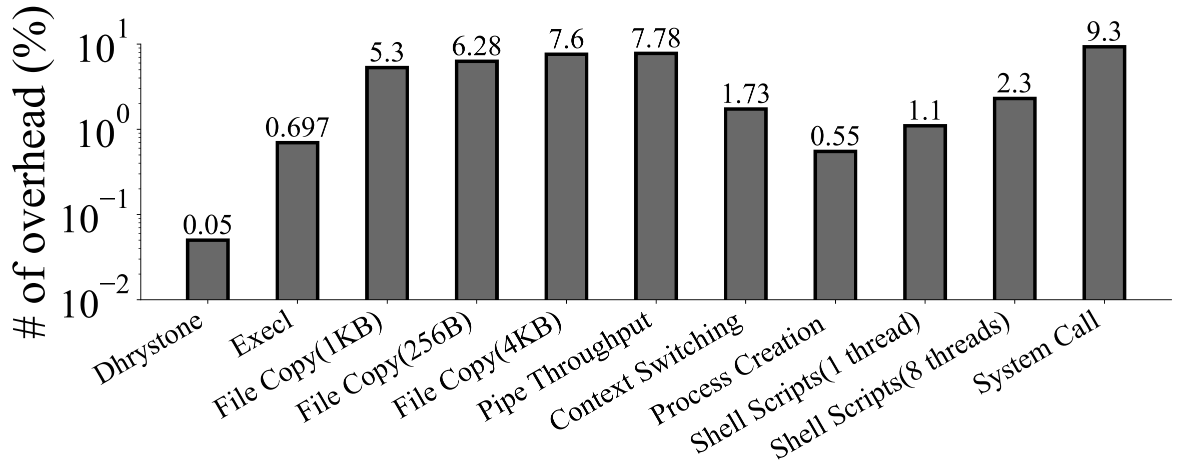
How is the overhead incurred by Investigator?

- The extra overhead comes from two aspects:
  ① tracing the executed instructions using ETM;
  ② retrieving the syscall data leveraging Capturer.

How is the overhead incurred by Investigator?

- The extra overhead comes from two aspects:
  ① tracing the executed instructions using ETM;
  ② retrieving the syscall data leveraging Capturer.

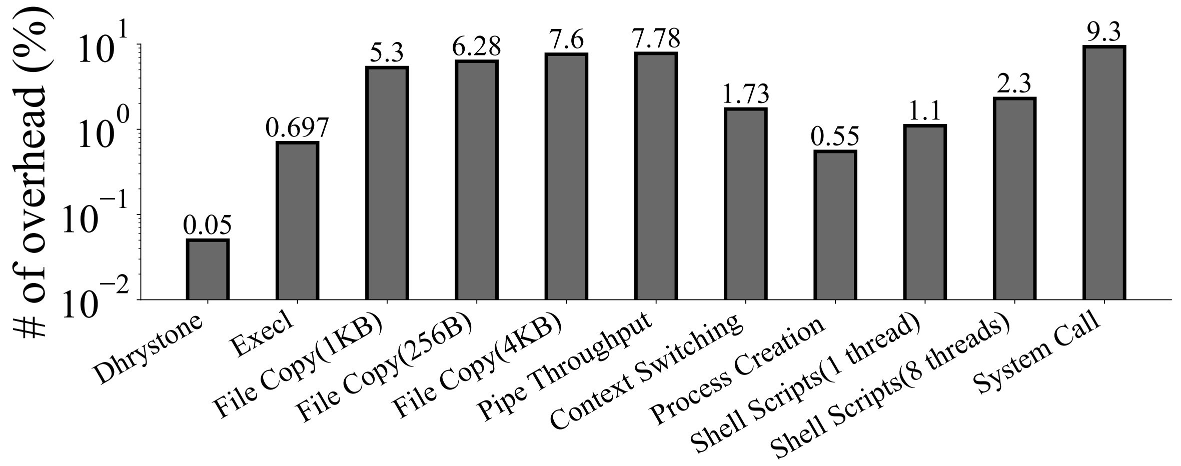
Table. Unixbench overhead incurred by Investigator.



How is the overhead incurred by Investigator?

- The extra overhead comes from two aspects:
  ① tracing the executed instructions using ETM;
  ② retrieving the syscall data leveraging Capturer.

Table. Unixbench overhead incurred by Investigator.

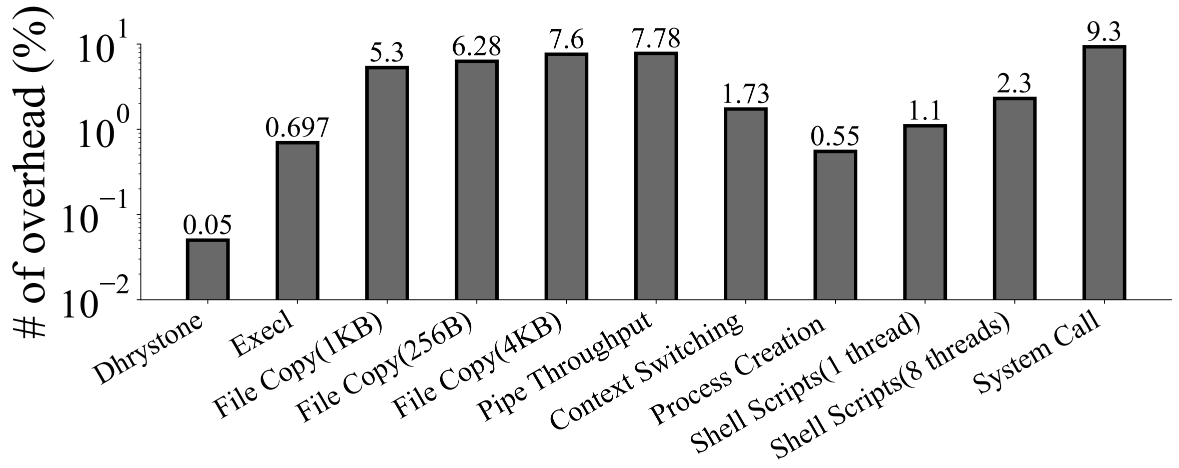


- Comparison between Investigator and REPT [6], a state-of-the-art diagnosis tool that is designed for in-production deployment on x86 architecture:
  - Investigator avg. 3.88%, highest 9.3%
  - REPT avg. 3.06%, highest 9.68%

How is the overhead incurred by Investigator?

- The extra overhead comes from two aspects:
  ① tracing the executed instructions using ETM;
  ② retrieving the syscall data leveraging Capturer.

Table. Unixbench overhead incurred by Investigator.



- Comparison between Investigator and REPT [6], a state-of-the-art diagnosis tool that is designed for in-production deployment on x86 architecture:
  - Investigator avg. 3.88%, highest 9.3%
  - REPT avg. 3.06%, highest 9.68%

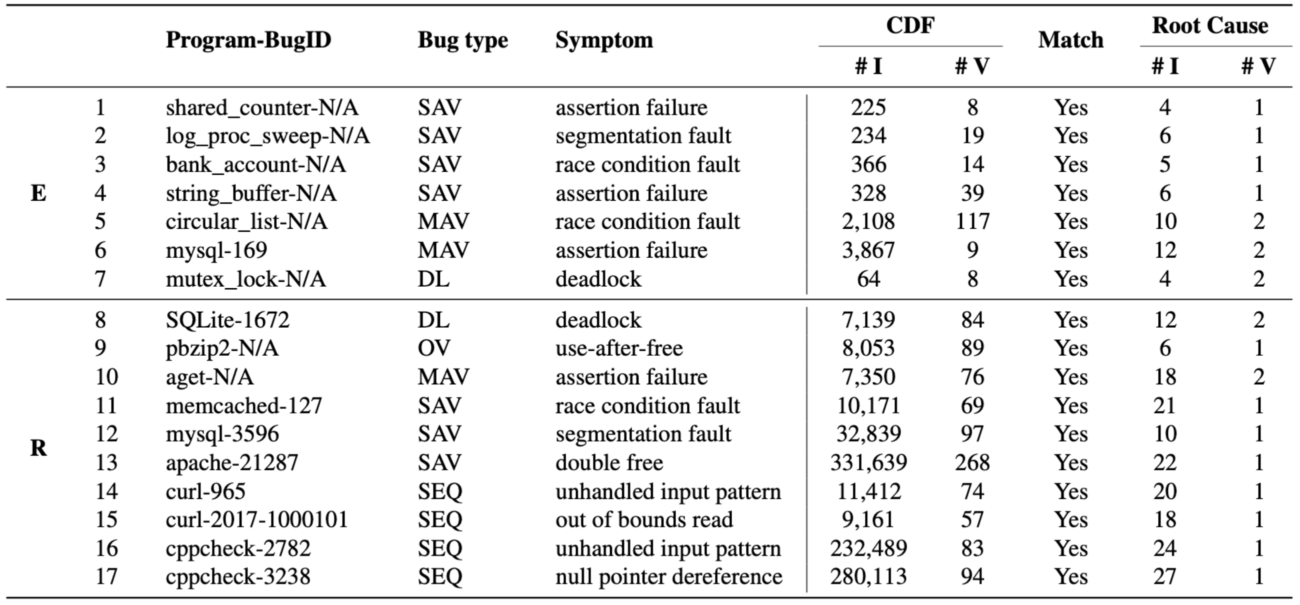
Comparable

Is the root cause diagnosing in Investigator effective?

Is the root cause diagnosing in Investigator effective?

- C/C++ buggy cases collected from bugbases

Table. Bugs diagnosed by Investigator.

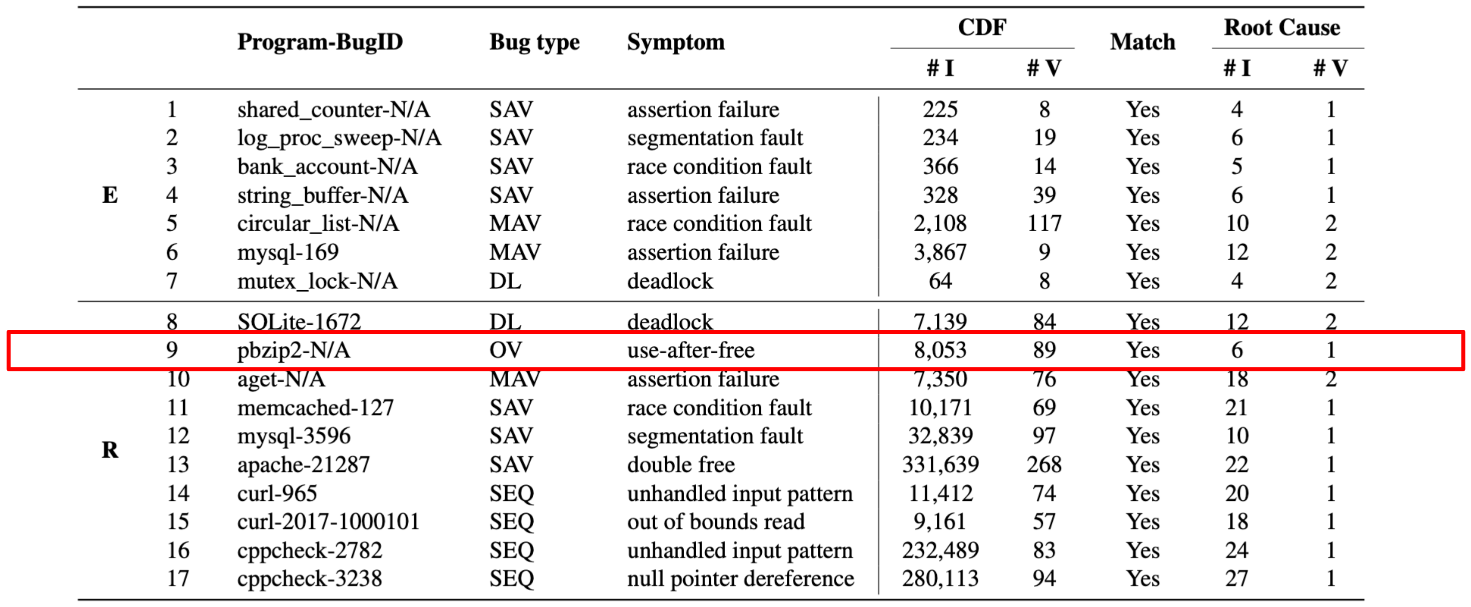


		Program-BugID	Bug type	Symptom	CDF		Match	Root Cause	
					# I	# V		# I	# V
E	1	shared_counter-N/A	SAV	assertion failure	225	8	Yes	4	1
	2	log_proc_sweep-N/A	SAV	segmentation fault	234	19	Yes	6	1
	3	bank_account-N/A	SAV	race condition fault	366	14	Yes	5	1
	4	string_buffer-N/A	SAV	assertion failure	328	39	Yes	6	1
	5	circular_list-N/A	MAV	race condition fault	2,108	117	Yes	10	2
	6	mysql-169	MAV	assertion failure	3,867	9	Yes	12	2
	7	mutex_lock-N/A	DL	deadlock	64	8	Yes	4	2
R	8	SQLite-1672	DL	deadlock	7,139	84	Yes	12	2
	9	pbzip2-N/A	OV	use-after-free	8,053	89	Yes	6	1
	10	aget-N/A	MAV	assertion failure	7,350	76	Yes	18	2
	11	memcached-127	SAV	race condition fault	10,171	69	Yes	21	1
	12	mysql-3596	SAV	segmentation fault	32,839	97	Yes	10	1
	13	apache-21287	SAV	double free	331,639	268	Yes	22	1
	14	curl-965	SEQ	unhandled input pattern	11,412	74	Yes	20	1
	15	curl-2017-1000101	SEQ	out of bounds read	9,161	57	Yes	18	1
	16	cppcheck-2782	SEQ	unhandled input pattern	232,489	83	Yes	24	1
	17	cppcheck-3238	SEQ	null pointer dereference	280,113	94	Yes	27	1

Is the root cause diagnosing in Investigator effective?

- C/C++ buggy cases collected from bugbases
  - Facilitation: a small number of instructions and variables from a large-scale control and data flow

Table. Bugs diagnosed by Investigator.

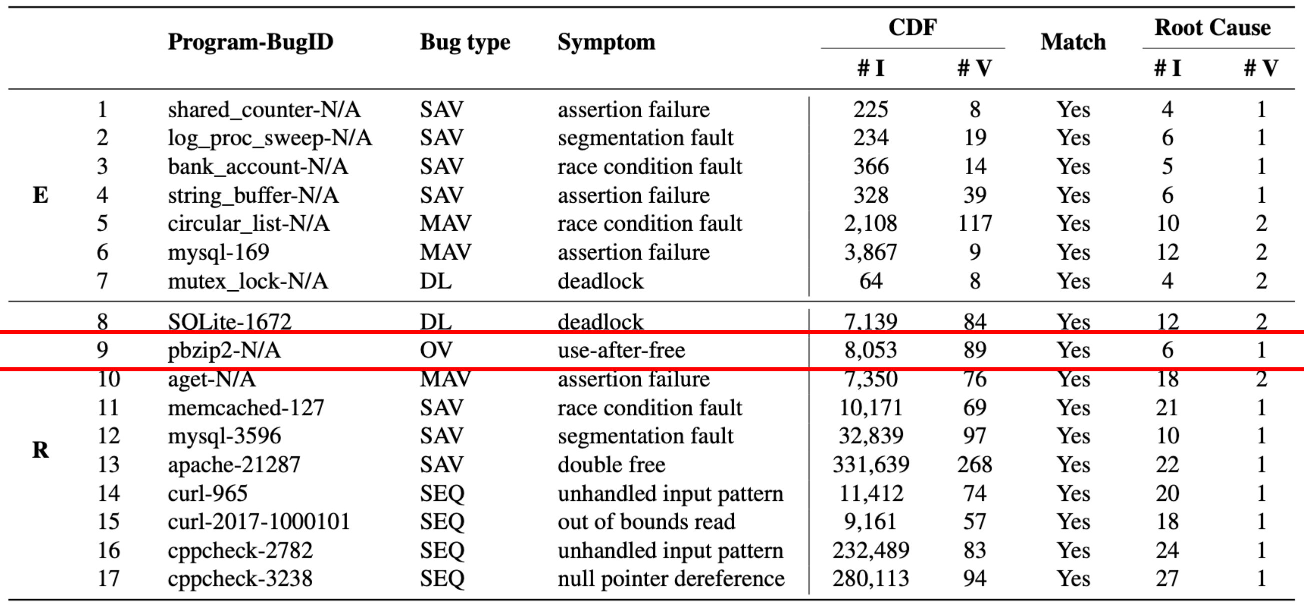


		Program-BugID	Bug type	Symptom	CDF		Match	Root Cause	
					# I	# V		# I	# V
E	1	shared_counter-N/A	SAV	assertion failure	225	8	Yes	4	1
	2	log_proc_sweep-N/A	SAV	segmentation fault	234	19	Yes	6	1
	3	bank_account-N/A	SAV	race condition fault	366	14	Yes	5	1
	4	string_buffer-N/A	SAV	assertion failure	328	39	Yes	6	1
	5	circular_list-N/A	MAV	race condition fault	2,108	117	Yes	10	2
	6	mysql-169	MAV	assertion failure	3,867	9	Yes	12	2
	7	mutex_lock-N/A	DL	deadlock	64	8	Yes	4	2
R	8	SQLite-1672	DL	deadlock	7,139	84	Yes	12	2
	9	pbzip2-N/A	OV	use-after-free	8,053	89	Yes	6	1
	10	aget-N/A	MAV	assertion failure	7,350	76	Yes	18	2
	11	memcached-127	SAV	race condition fault	10,171	69	Yes	21	1
	12	mysql-3596	SAV	segmentation fault	32,839	97	Yes	10	1
	13	apache-21287	SAV	double free	331,639	268	Yes	22	1
	14	curl-965	SEQ	unhandled input pattern	11,412	74	Yes	20	1
	15	curl-2017-1000101	SEQ	out of bounds read	9,161	57	Yes	18	1
	16	cppcheck-2782	SEQ	unhandled input pattern	232,489	83	Yes	24	1
	17	cppcheck-3238	SEQ	null pointer dereference	280,113	94	Yes	27	1

Is the root cause diagnosing in Investigator effective?

- C/C++ buggy cases collected from bugbases
  - Facilitation: a small number of instructions and variables from a large-scale control and data flow

Table. Bugs diagnosed by Investigator.



		Program-BugID	Bug type	Symptom	CDF		Match	Root Cause	
					# I	# V		# I	# V
E	1	shared_counter-N/A	SAV	assertion failure	225	8	Yes	4	1
	2	log_proc_sweep-N/A	SAV	segmentation fault	234	19	Yes	6	1
	3	bank_account-N/A	SAV	race condition fault	366	14	Yes	5	1
	4	string_buffer-N/A	SAV	assertion failure	328	39	Yes	6	1
	5	circular_list-N/A	MAV	race condition fault	2,108	117	Yes	10	2
	6	mysql-169	MAV	assertion failure	3,867	9	Yes	12	2
	7	mutex_lock-N/A	DL	deadlock	64	8	Yes	4	2
R	8	SQLite-1672	DL	deadlock	7,139	84	Yes	12	2
	9	pbzip2-N/A	OV	use-after-free	8,053	89	Yes	6	1
	10	aget-N/A	MAV	assertion failure	7,350	76	Yes	18	2
	11	memcached-127	SAV	race condition fault	10,171	69	Yes	21	1
	12	mysql-3596	SAV	segmentation fault	32,839	97	Yes	10	1
	13	apache-21287	SAV	double free	331,639	268	Yes	22	1
	14	curl-965	SEQ	unhandled input pattern	11,412	74	Yes	20	1
	15	curl-2017-1000101	SEQ	out of bounds read	9,161	57	Yes	18	1
	16	cppcheck-2782	SEQ	unhandled input pattern	232,489	83	Yes	24	1
	17	cppcheck-3238	SEQ	null pointer dereference	280,113	94	Yes	27	1

- Effectiveness: Patches indicating the location that the developers fix the bug match our diagnosis results

Summary and Future Work

Summary and Future Work

- Investigator: a new hardware-assisted framework on Arm for failure diagnosis in production.

Summary and Future Work

- Investigator: a new hardware-assisted framework on Arm for failure diagnosis in production.
- Designing methods to record execution pertaining to failures with low overhead.

Summary and Future Work

- Investigator: a new hardware-assisted framework on Arm for failure diagnosis in production.
- Designing methods to record execution pertaining to failures with low overhead.
- Conducting accurately execution flow recovery, which provides developers with sufficient information for root cause analysis.

Summary and Future Work

- Investigator: a new hardware-assisted framework on Arm for failure diagnosis in production.
- Designing methods to record execution pertaining to failures with low overhead.
- Conducting accurately execution flow recovery, which provides developers with sufficient information for root cause analysis.

❖ Extend Investigator to support other root cause diagnosis methods

Thanks for listening!

Q & A

Reference

- [1] Zuo G, Ma J, Quinn A, et al. “Execution reconstruction: Harnessing failure reoccurrences for failure reproduction” Proc. PLDI. 2021
- [2] Jun Xu, Dongliang Mu, et al. “Postmortem program analysis with hardware-enhanced post-crash artifacts”. Proc. USENIX Security. 2017.
- [3] Zhenyu, and Fengwei Zhang. “Ninja: Towards Transparent Tracing and Debugging on ARM”. Proc. USENIX Security. 2017.
- [4] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. Ph. D. Dissertation. Citeseer.
- [5] Baris Kasikci, Weidong Cui, et al. 2017. “Lazy Diagnosis of In-Production Concurrency Bugs”. Proc. SOSP. 2017.
- [6] Xinyang Ge, Ben Niu, et al. 2020. “Reverse debugging of kernel failures in deployed systems”. Proc. USENIX ATC . 2020.