ACM Conference on Computer and Communications Security (CCS) 2023

Whole-Program Control-flow Path Attestation

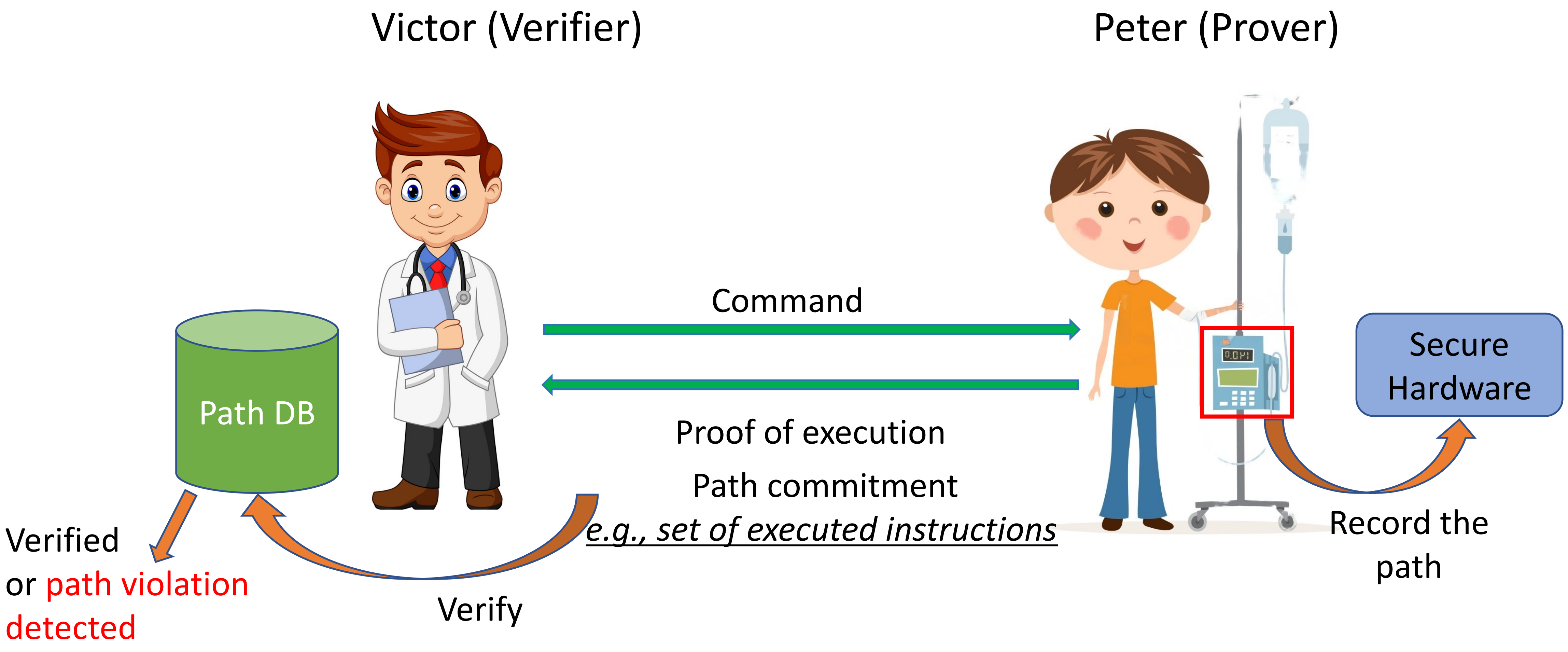
Nikita Yadav and Vinod Ganapathy

CSSL Computer Systems Security Laboratory
Indian Institute of Science, Bangalore

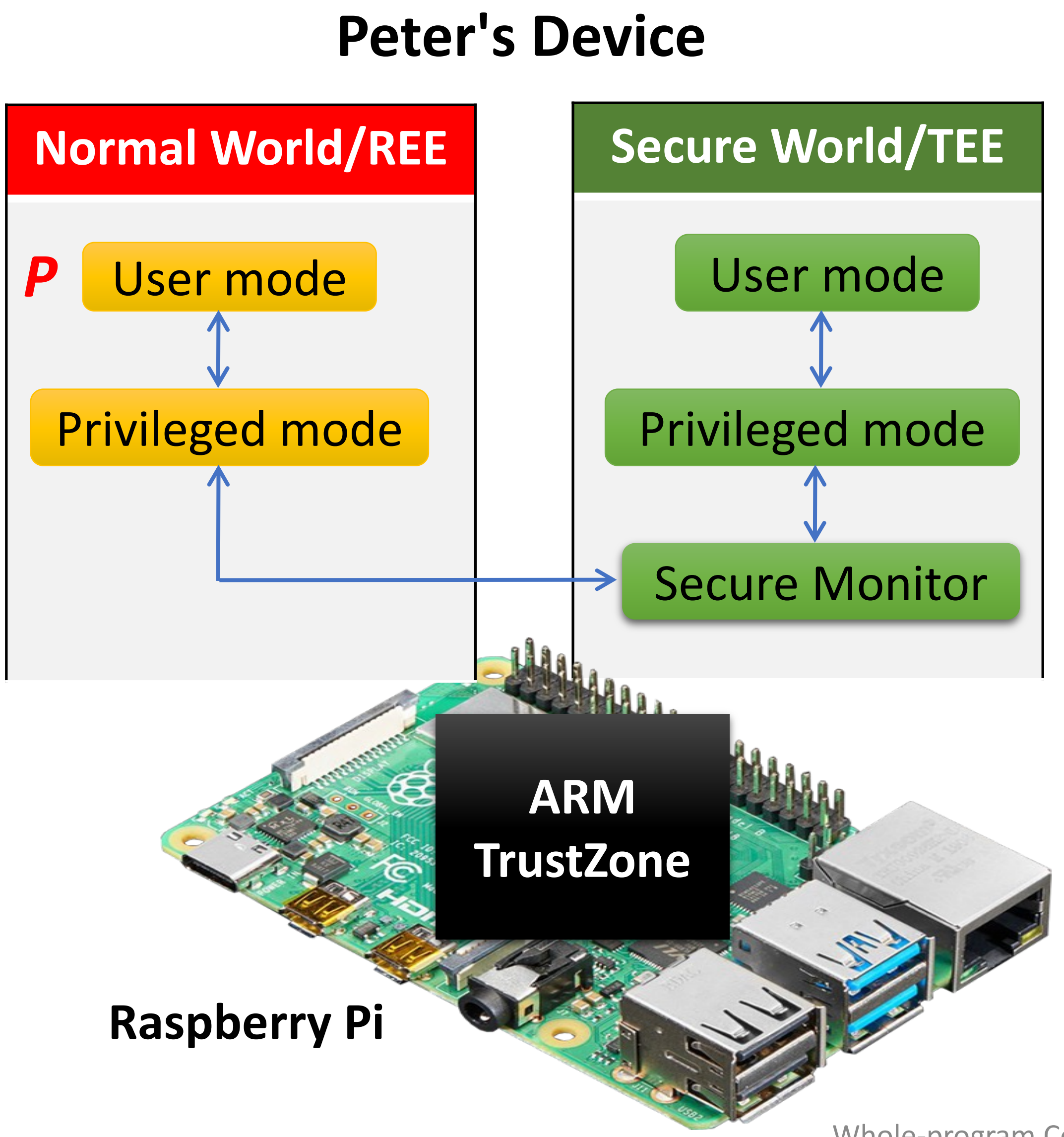
INDIAN INSTITUTE OF SCIENCE
भारतीय विज्ञान संस्थान

nikitayadav@iisc.ac.in, vg@iisc.ac.in

Problem Setting



Background & Threat Model



Capabilities of TEE:

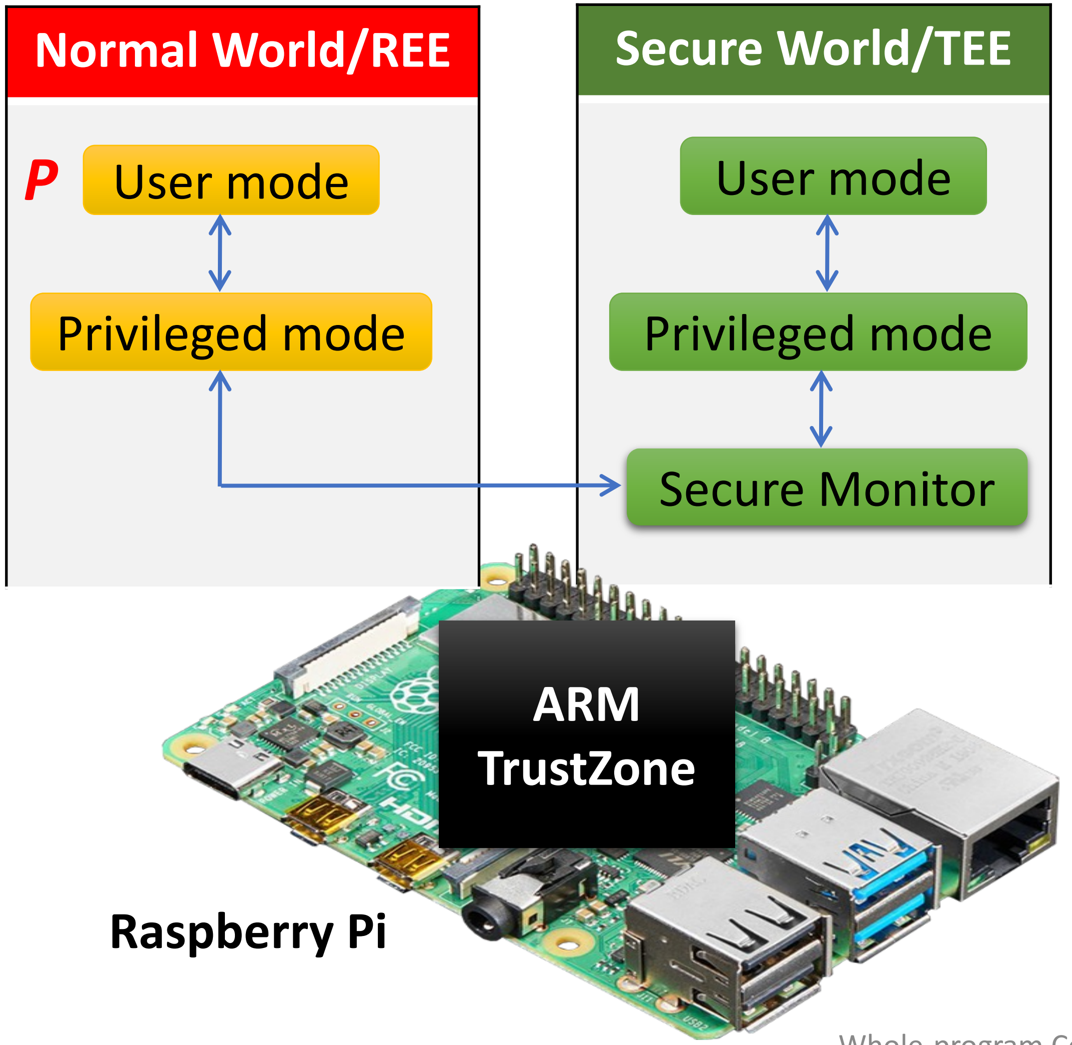
1. Verify REE configuration.
2. Generate digital signatures.
3. Provides secure storage.

Assumptions:

1. TEE is available.
2. Data Execution Prevention (DEP) is enabled by REE OS, attested by TEE.

Background & Threat Model

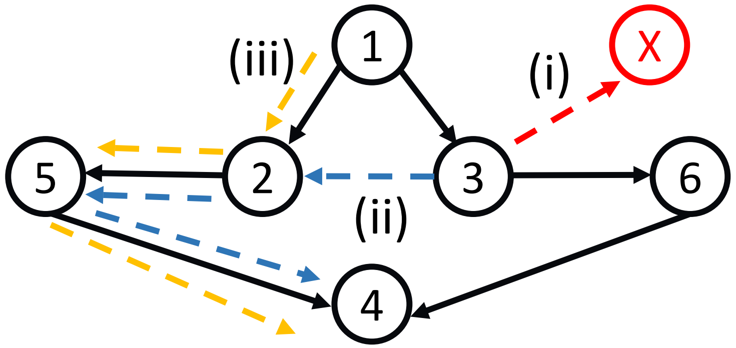
Peter's Device



Possible Threats:

1. ***P*** could be modified
2. Code injection in ***P***
3. Code-reuse attacks/ Return-oriented attacks.
4. Input corruption/Data corruption
5. Out of scope – Physical attacks.

Runtime Attacks



Types of Runtime attacks

(i) Attacker injected code execution

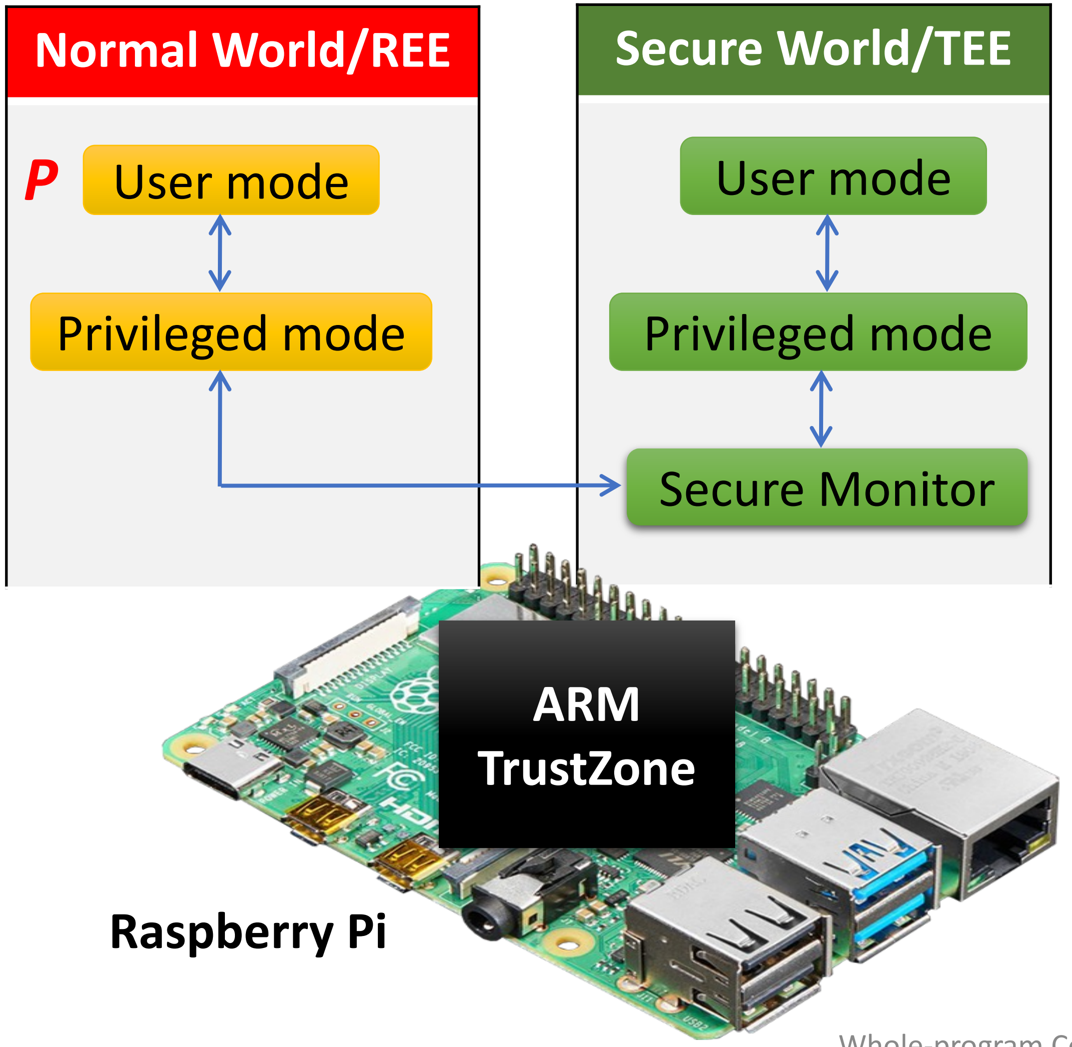
(ii) Code-reuse attack

(iii) Non-control data attack

Source: CFLAT – Control-Flow Attestation for Embedded System Software, CCS'16

Background & Threat Model

Peter's Device



Raspberry Pi

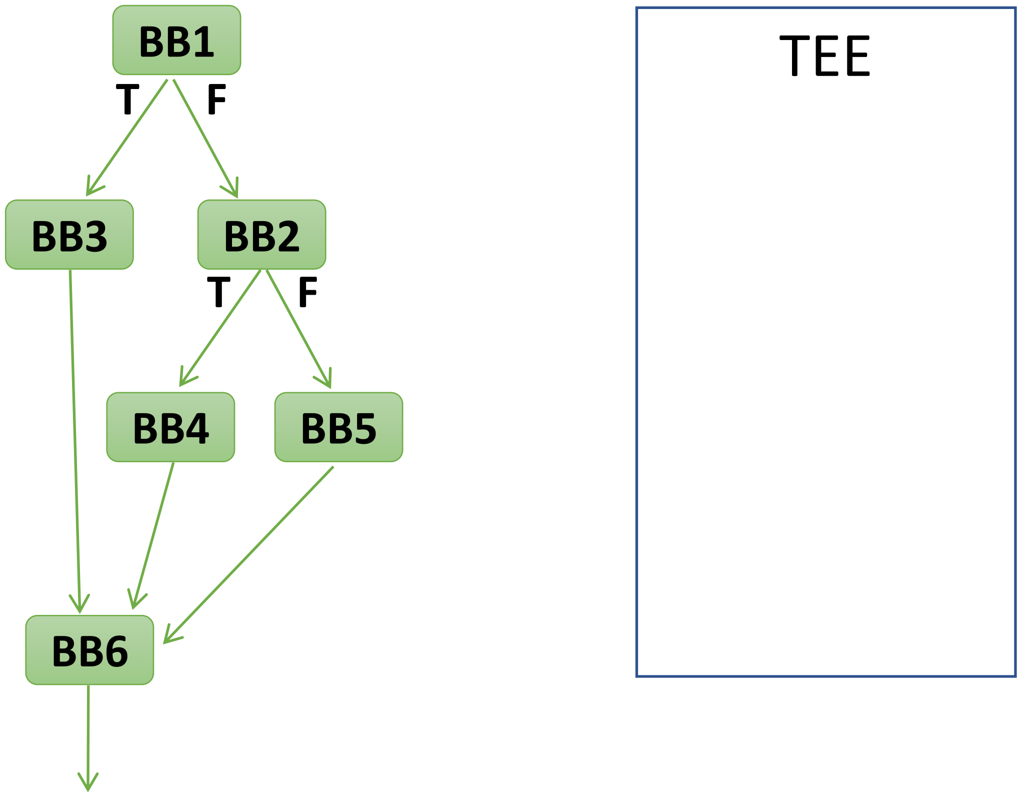
Possible Threats:

1. ***P*** could be modified -> TEE attests the code image of P in REE.
2. Code injection in ***P*** -> DEP, ensured by TEE attestation of REE OS.
3. Code-reuse attacks/ Return-oriented attacks. -> This work
4. Input corruption/Data corruption -> This work
5. Out of scope – Physical attacks.

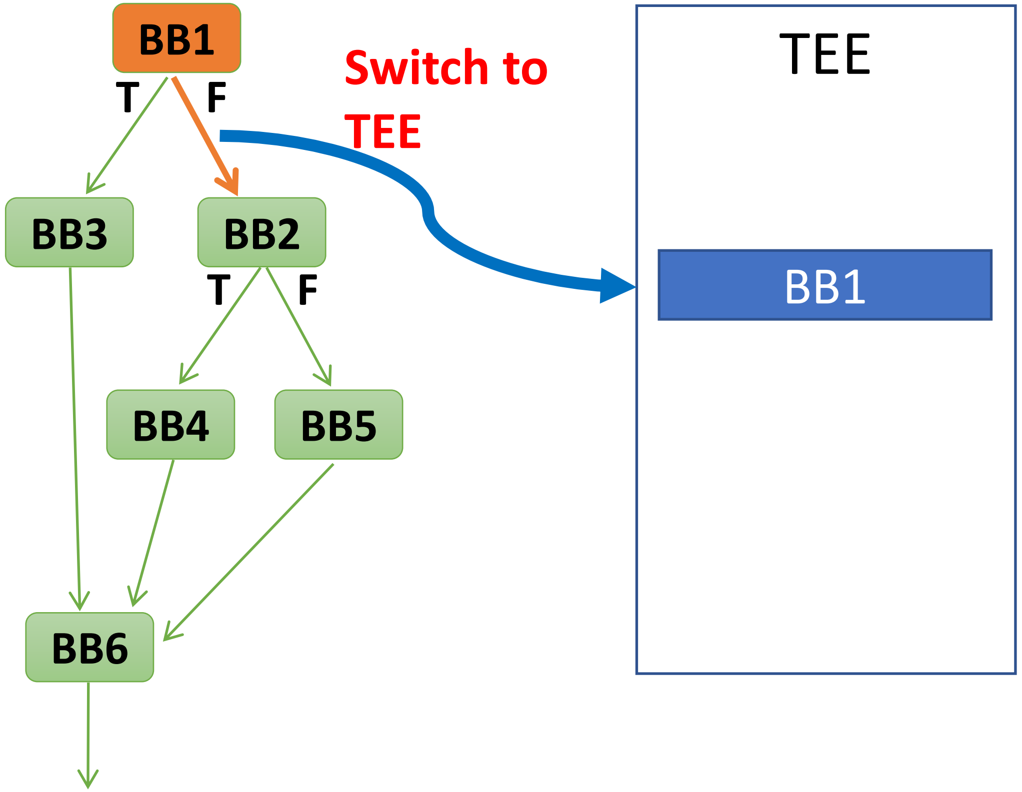
Problem

Record program execution path securely.

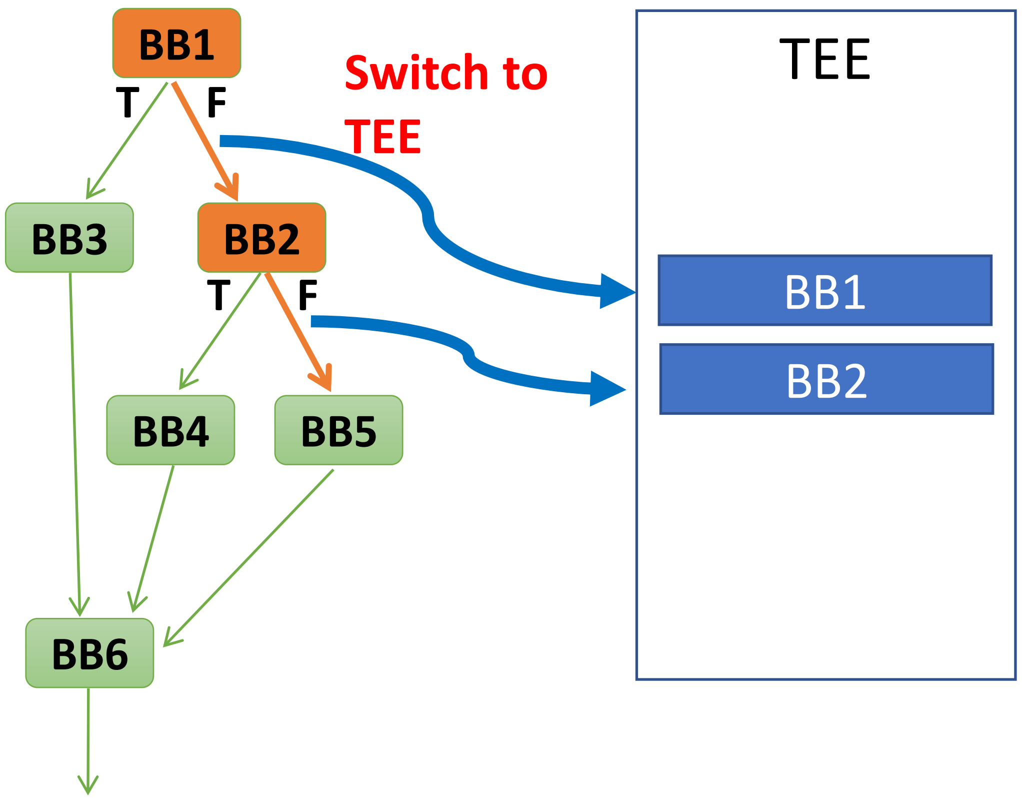
Strawman Approach I



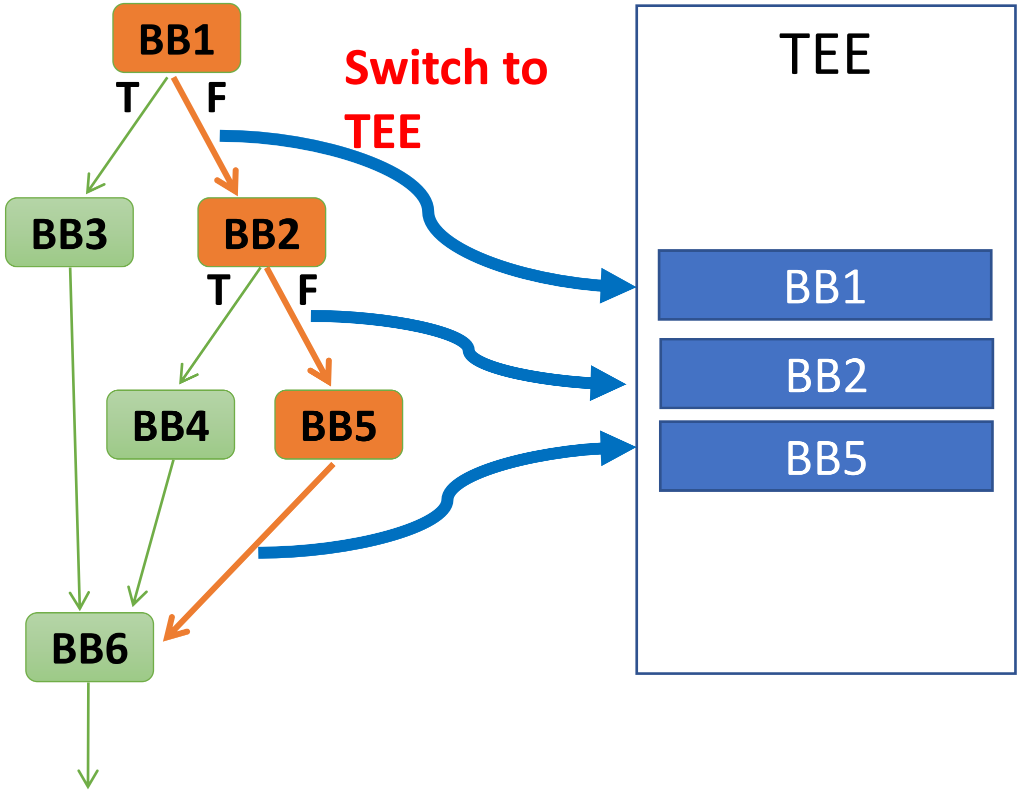
Strawman Approach I



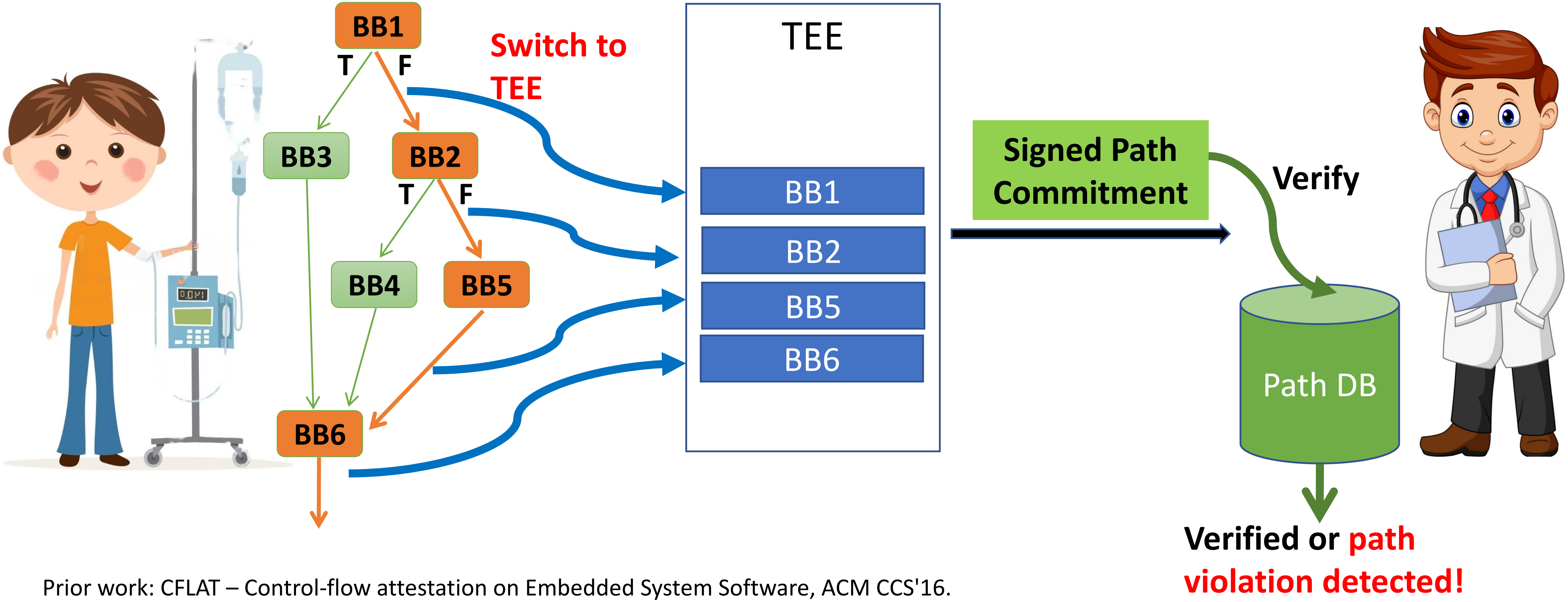
Strawman Approach I



Strawman Approach I

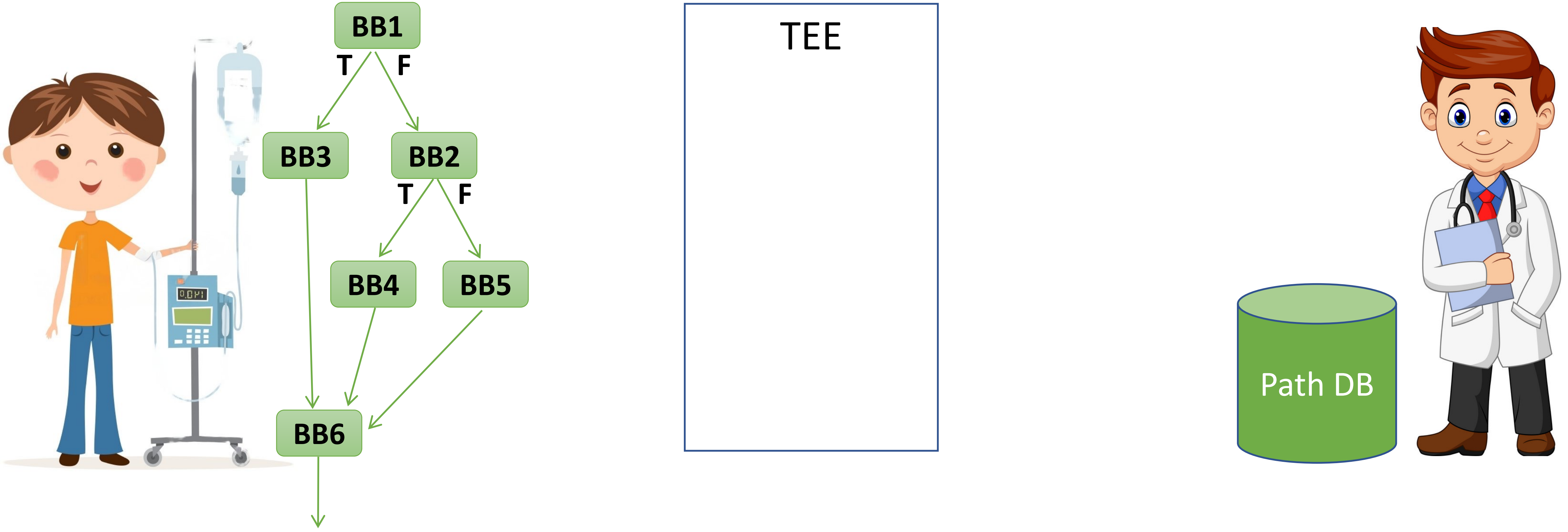


Strawman Approach I

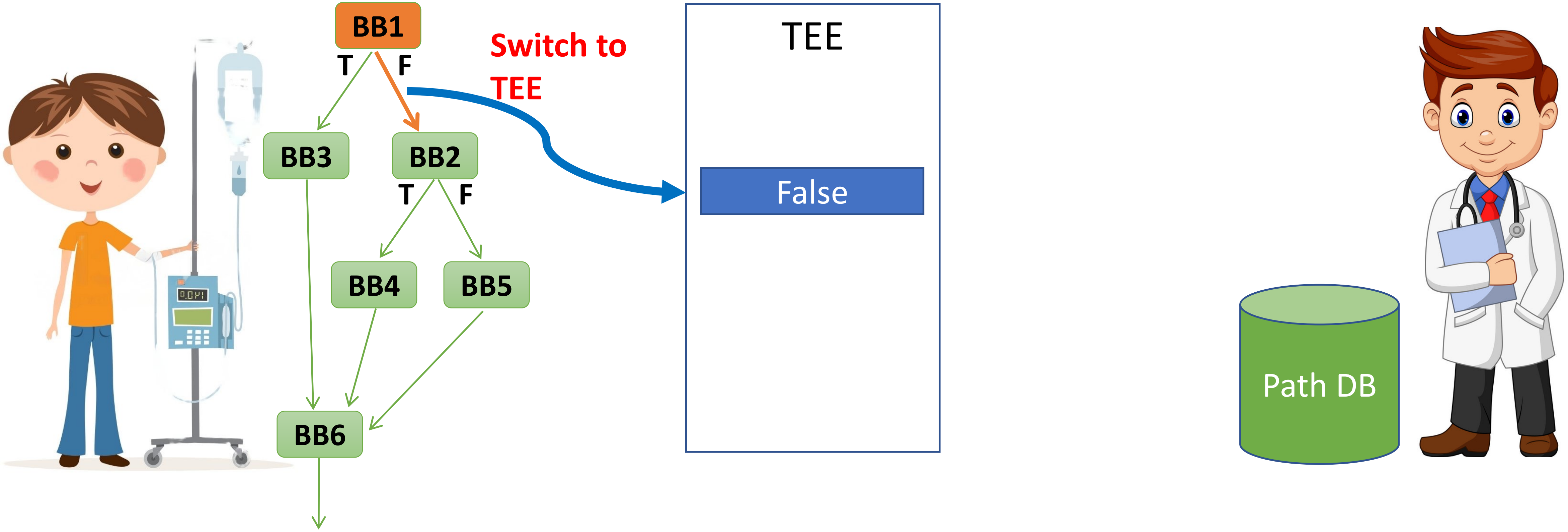


Prior work: CFLAT – Control-flow attestation on Embedded System Software, ACM CCS'16.

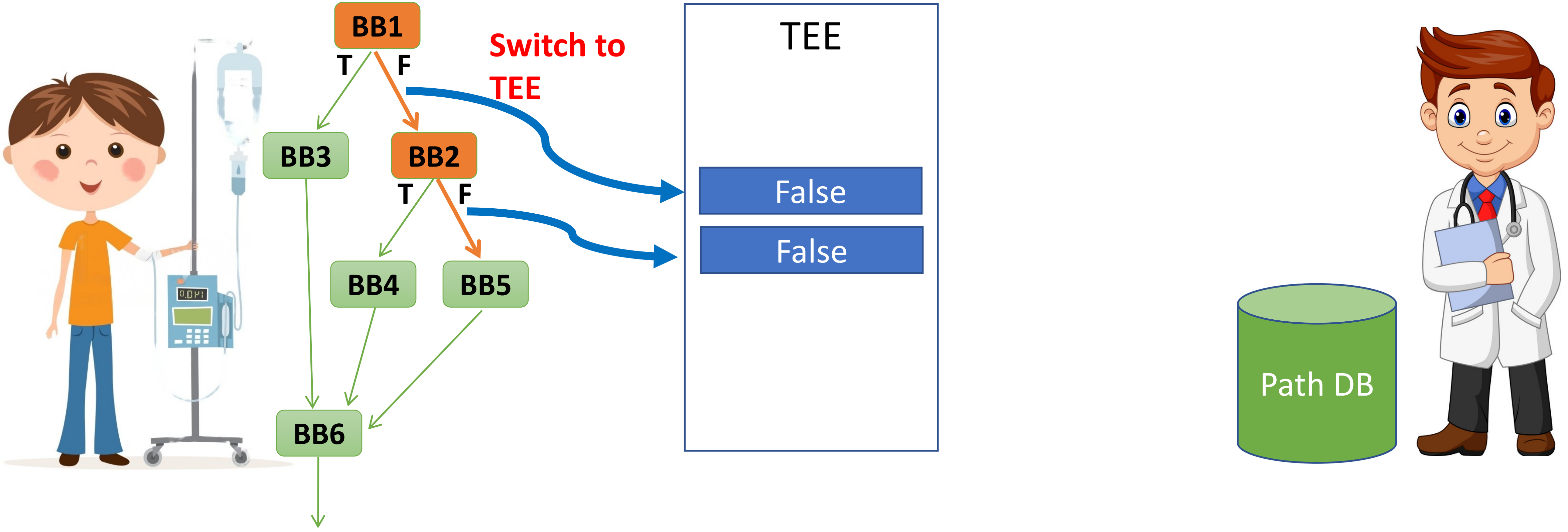
Strawman Approach II



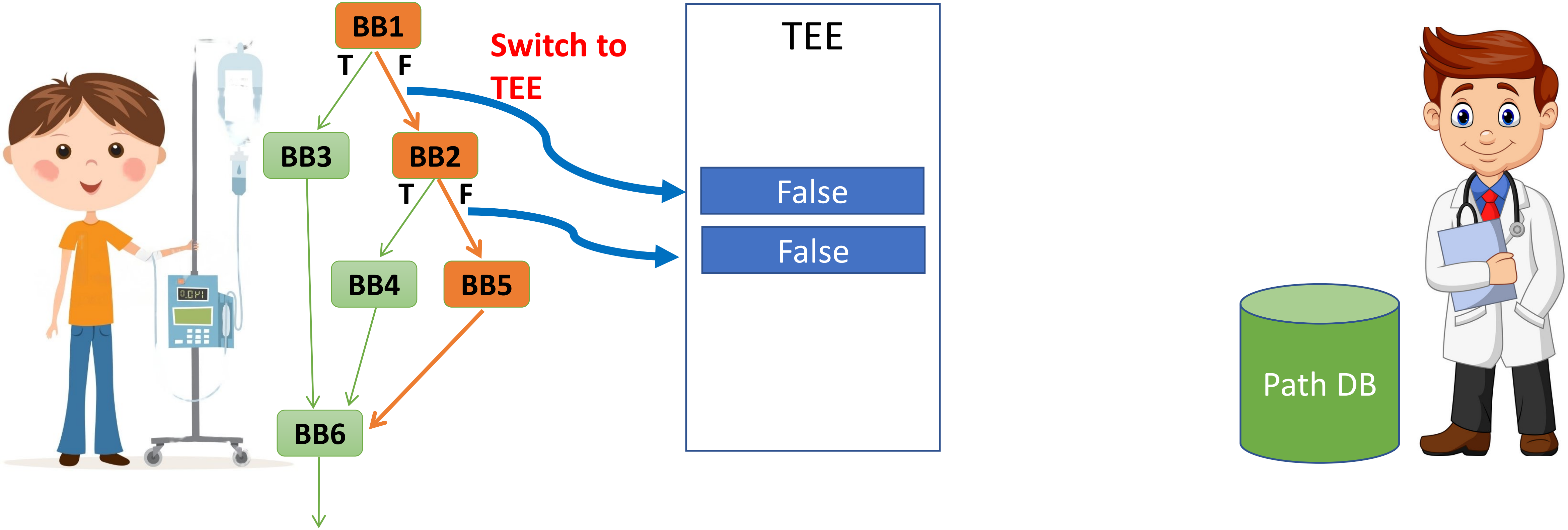
Strawman Approach II



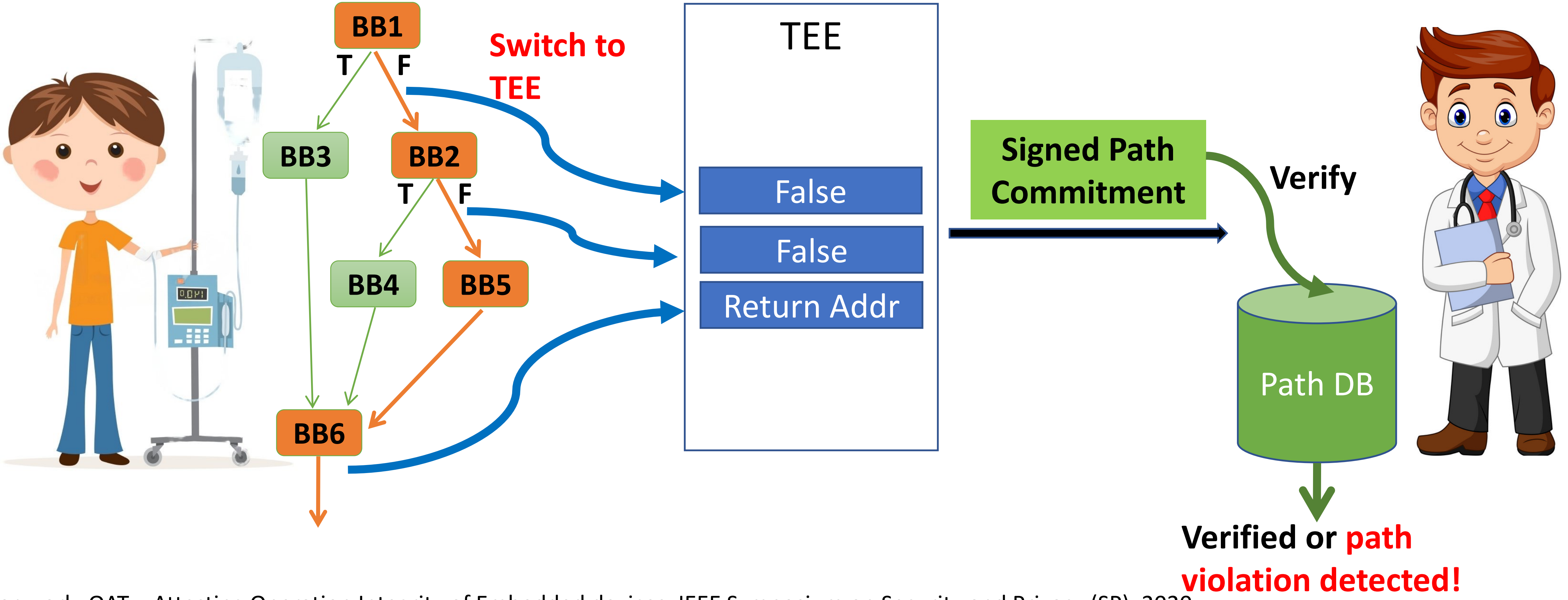
Strawman Approach II



Strawman Approach II



Strawman Approach II



Prior work: OAT – Attesting Operation Integrity of Embedded devices, IEEE Symposium on Security and Privacy (SP), 2020

Overhead Reports by CFLAT & OAT

CFLAT reported 0.13 % overhead for syringe pump benchmark.

OAT reported an average overhead of 2.7% on five embedded programs.

Evaluation on Embench-IoT Benchmark

Embench-IoT Benchmark	Total TEE domain Switches Encountered at Runtime	
Program ⇩	Strawman Approach I (CFLAT)	Strawman Approach II (OAT)
aha-mont64	857,844,016	392,967,008
crc32	871,930,016	348,840,008
cubic	2,030,022	860,013
edn	1,106,118,020	372,621,011
huffbench	984,236,016	496,903,008
matmul-int	1,201,018,222	406,825,691
minver	277,500,079	115,440,042
nbody	17,279,126	6,329,070
nettle-aes	227,449,298	78,858,777
nettle-sha256	223,250,050	34,200,025
primecount	1,607,180,016	880,206,008

Effect of TEE switches on Runtime

Embench-IoT Benchmark	Total TEE domain Switches Encountered at Runtime	
Program ⇩	CFLAT	OAT
nettle-sha256	223,250,050	34,200,025

1 TEE domain switch takes ~ 190 µsecs on Raspberry Pi.

X 190 µsecs **X 190 µsecs**

Baseline Execution Time	Time with CFLAT	Time with OAT
12 seconds	**> 11 hours**	**~ 2 hours**

CFLAT and OAT impose over 1000X Overhead on all Benchmarks due to high number of TEE domain switches.

Rationale for low overhead of CFLAT & OAT

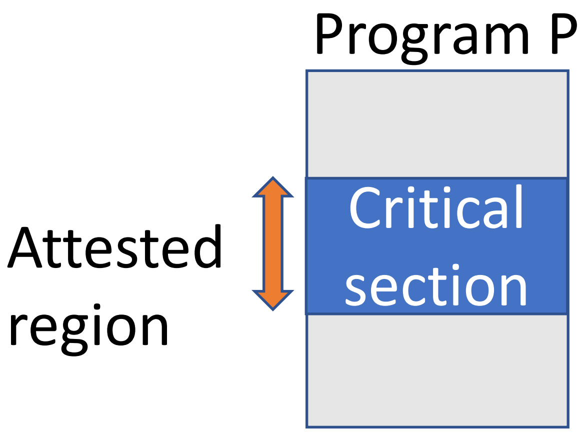
I. Prior works evaluate **small embedded programs** with only few hundreds of control-flow events.

II. Attest **only critical sections** of the program (CFLAT) or certain operations in the program (OAT).

Rationale for low overhead of CFLAT & OAT

I. Prior works evaluate **small embedded programs** with only few hundreds of control-flow events.

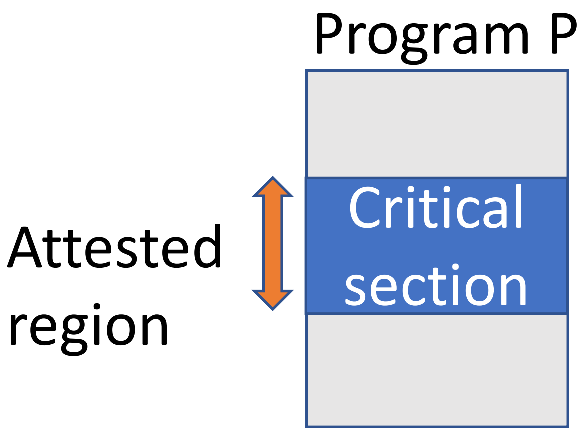
II. Attest **only critical sections** of the program (CFLAT) or certain operations in the program (OAT).

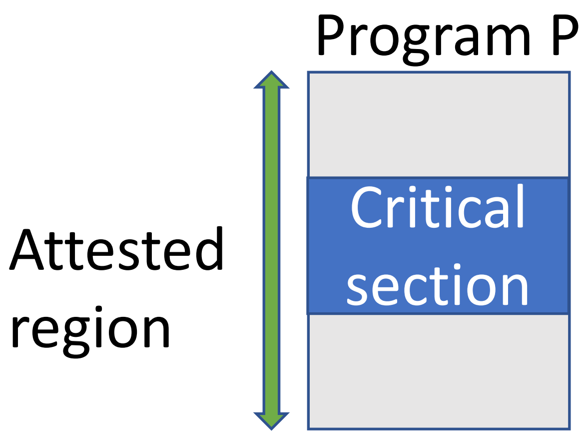


Rationale for low overhead of CFLAT & OAT

I. Prior works evaluate **small embedded programs** with only few hundreds of control-flow events. **-> This work evaluate on Embench-IoT benchmark.**

II. Attest **only critical sections** of the program (CFLAT) or certain operations in the program (OAT). **-> This work attests whole-programs.**





Ref: A Probability Prediction Based Mutable Control-Flow Attestation Scheme on Embedded Platforms

Selective Attestation

```
void func1( ) {
1     .....
2     scanf("%d", &n)   ←   input: 5
3     ........
4     if (flag>0) func2( );
5     else func3(n);
6 }

  void func2( ) {
     ....
  }
```

```
void func3(int n ) {
7     ......      →   n = 3
8  while (n) {
9        n- = 1;
   }
}
```

Attack is missed when only func1 and func2 are attested and not func3.

Ref: A Probability Prediction Based Mutable Control-Flow Attestation Scheme on Embedded Platforms

Conclusion

State-of-the-art path attestation approaches are extremely slow and attests only parts of the program.

BLAST

Whole-program path attestation with near-practical overhead.

Key Contributions

1) Store path locally in log (reduces TEE domain switches)

2) Instrument *P* using Ball Larus Profiling (reduces log entries)

3) Compact & expressive path representation

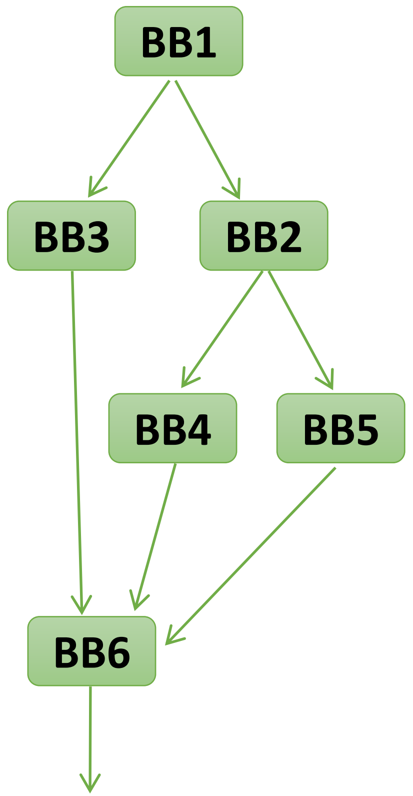
Key Contributions

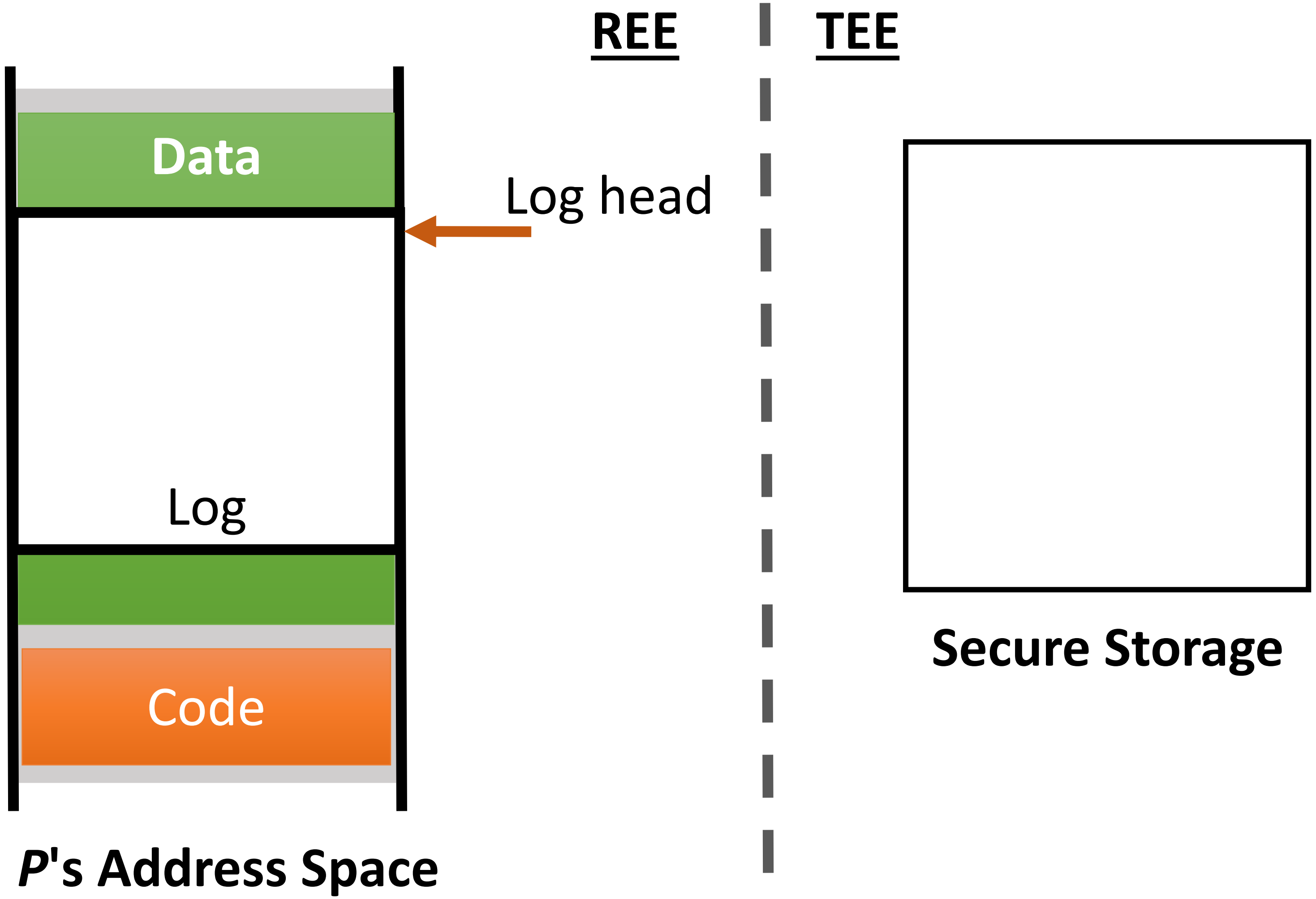
1) Store path locally in log (reduces TEE domain switches)

2) Instrument *P* using Ball Larus Profiling (reduces log entries)

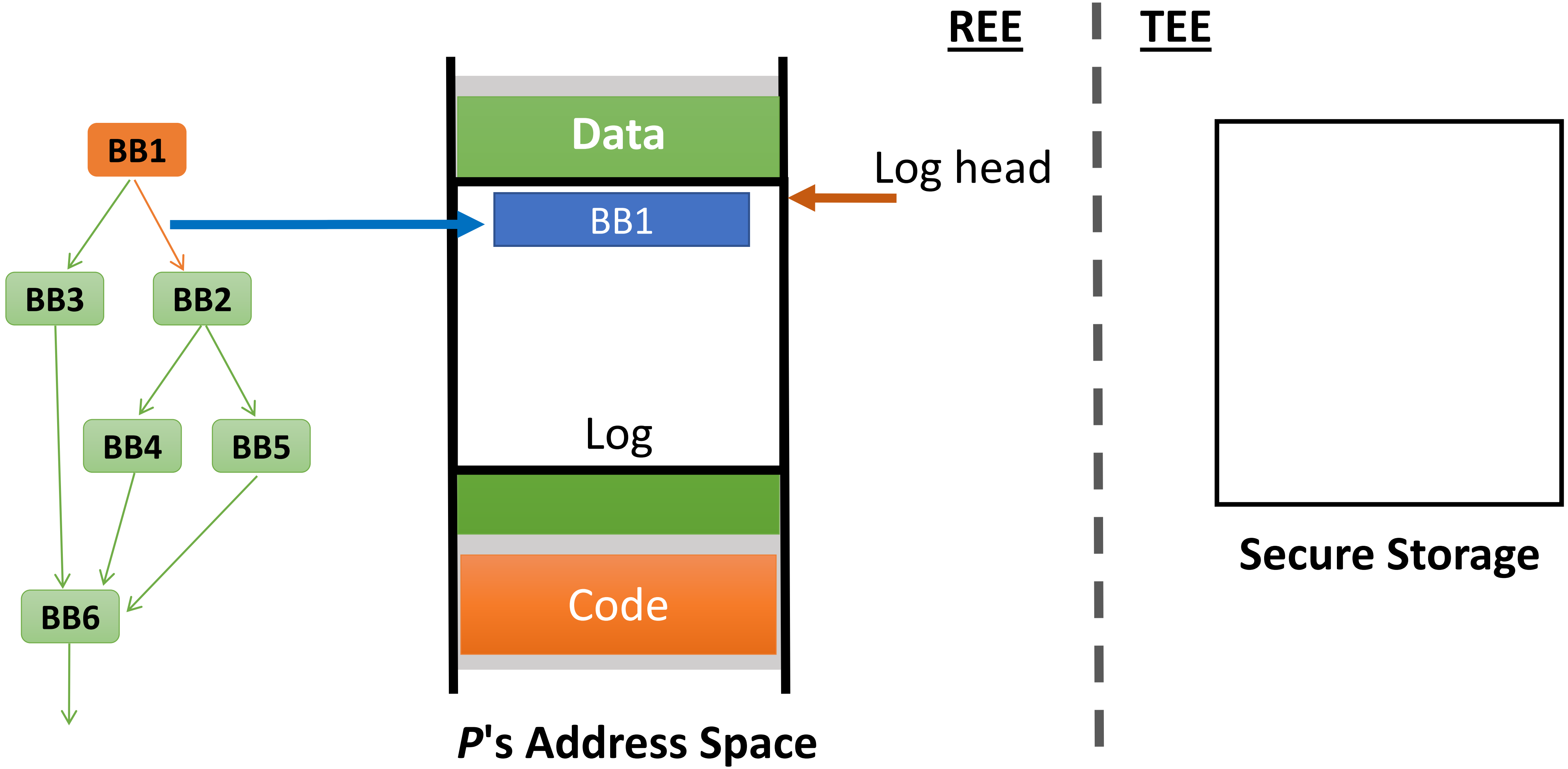
3) Compact & expressive path representation

Buffer in Local Log

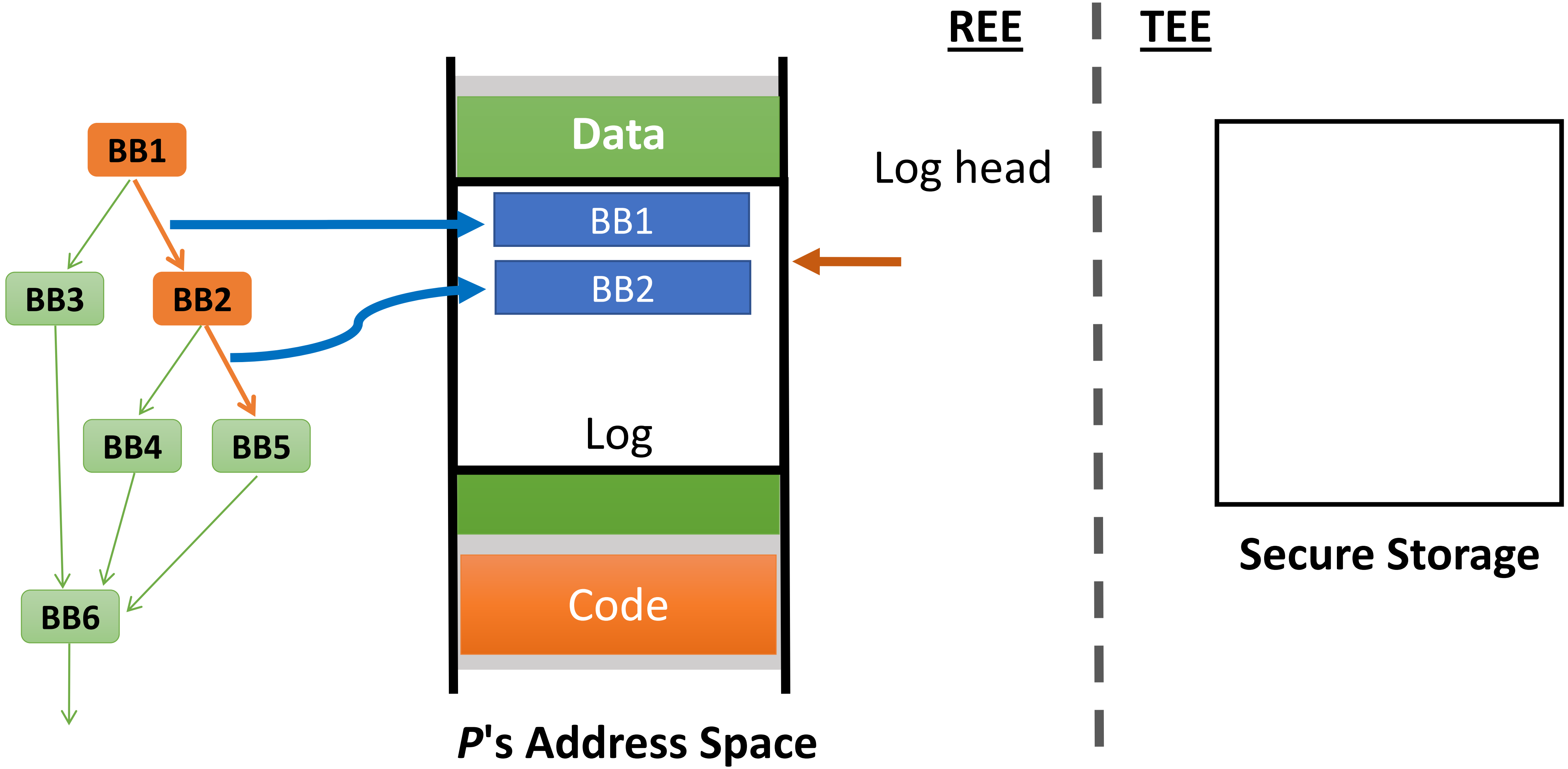




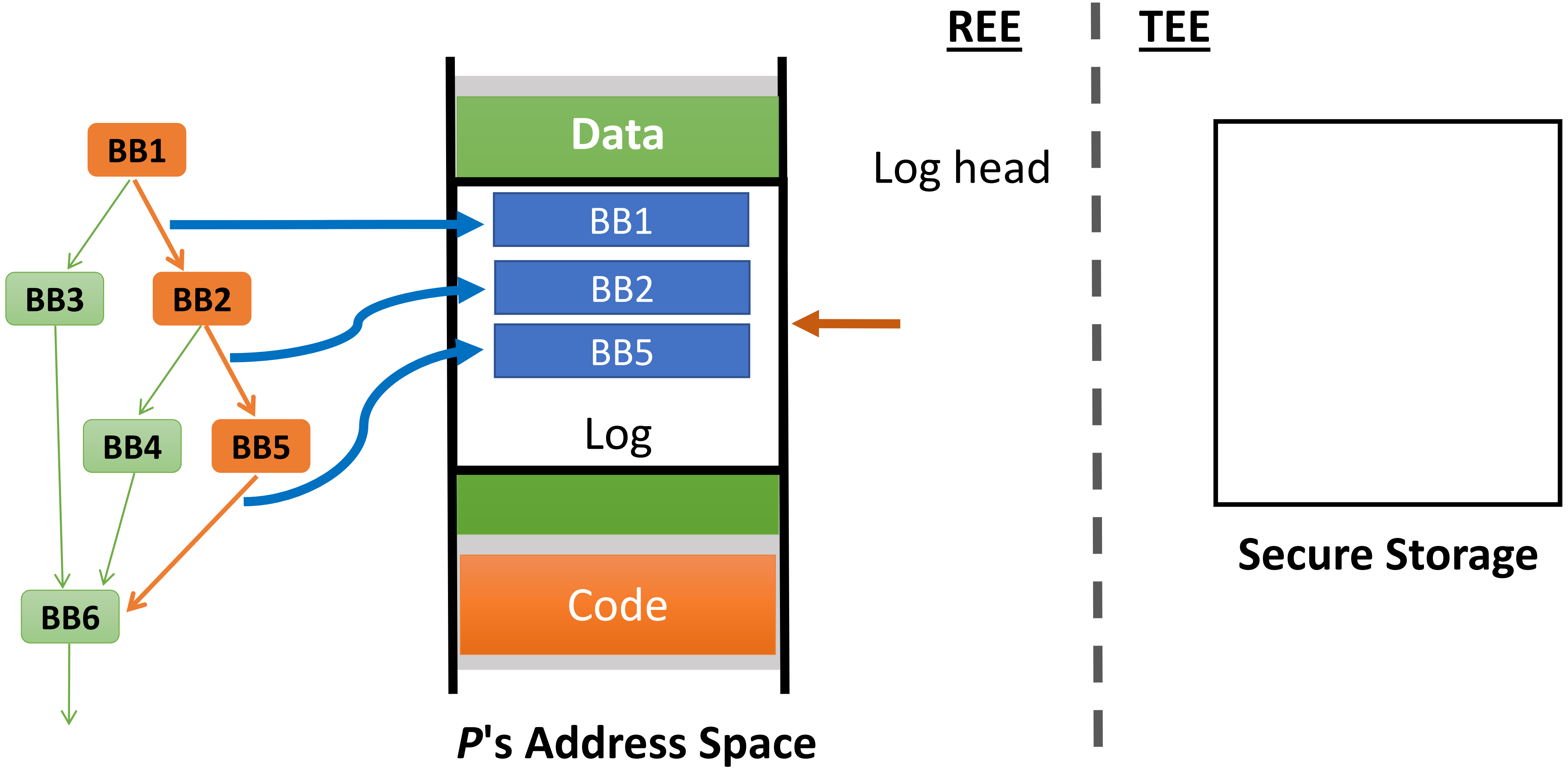
Buffer in Local Log



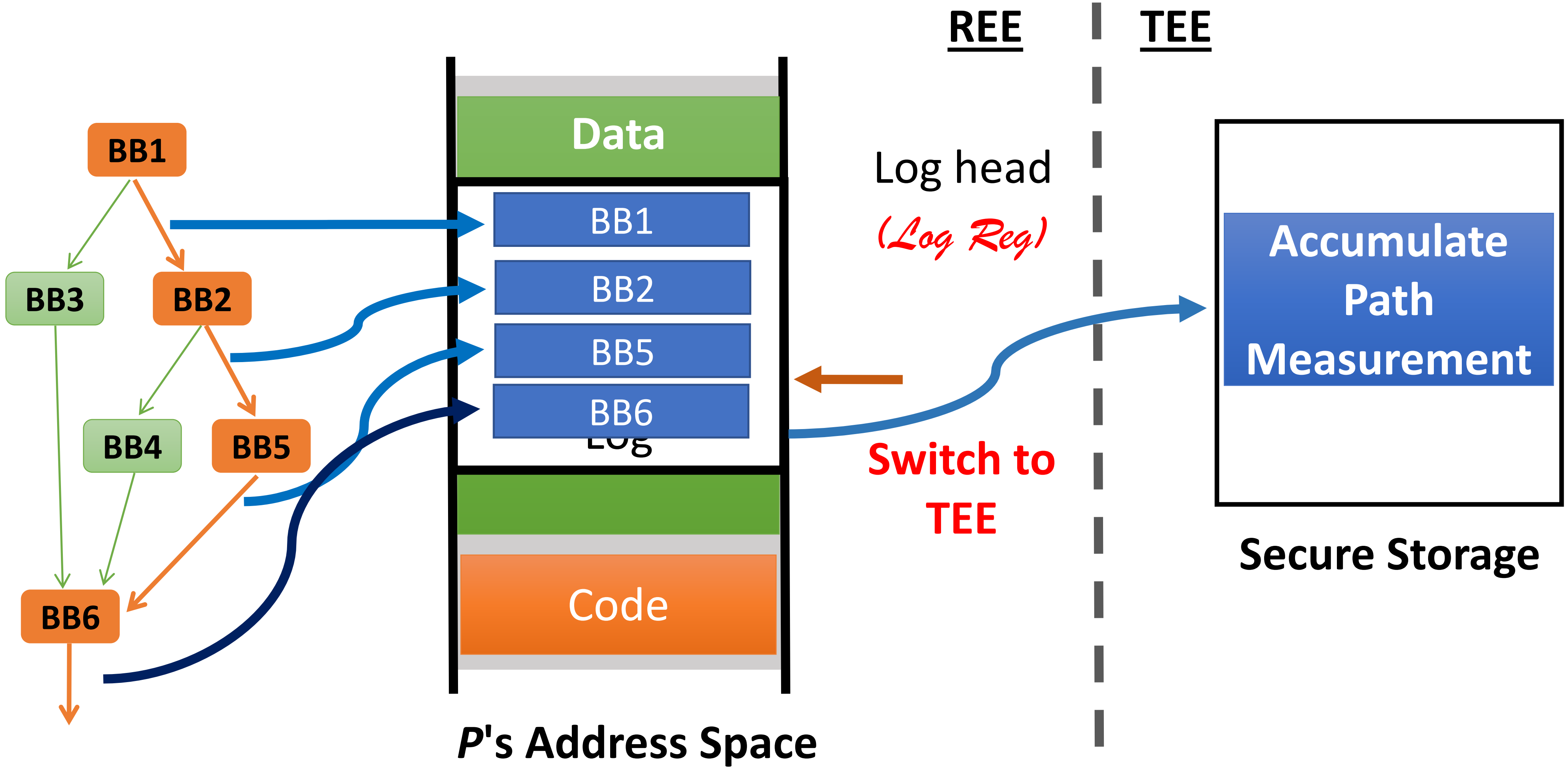
Buffer in Local Log



Buffer in Local Log

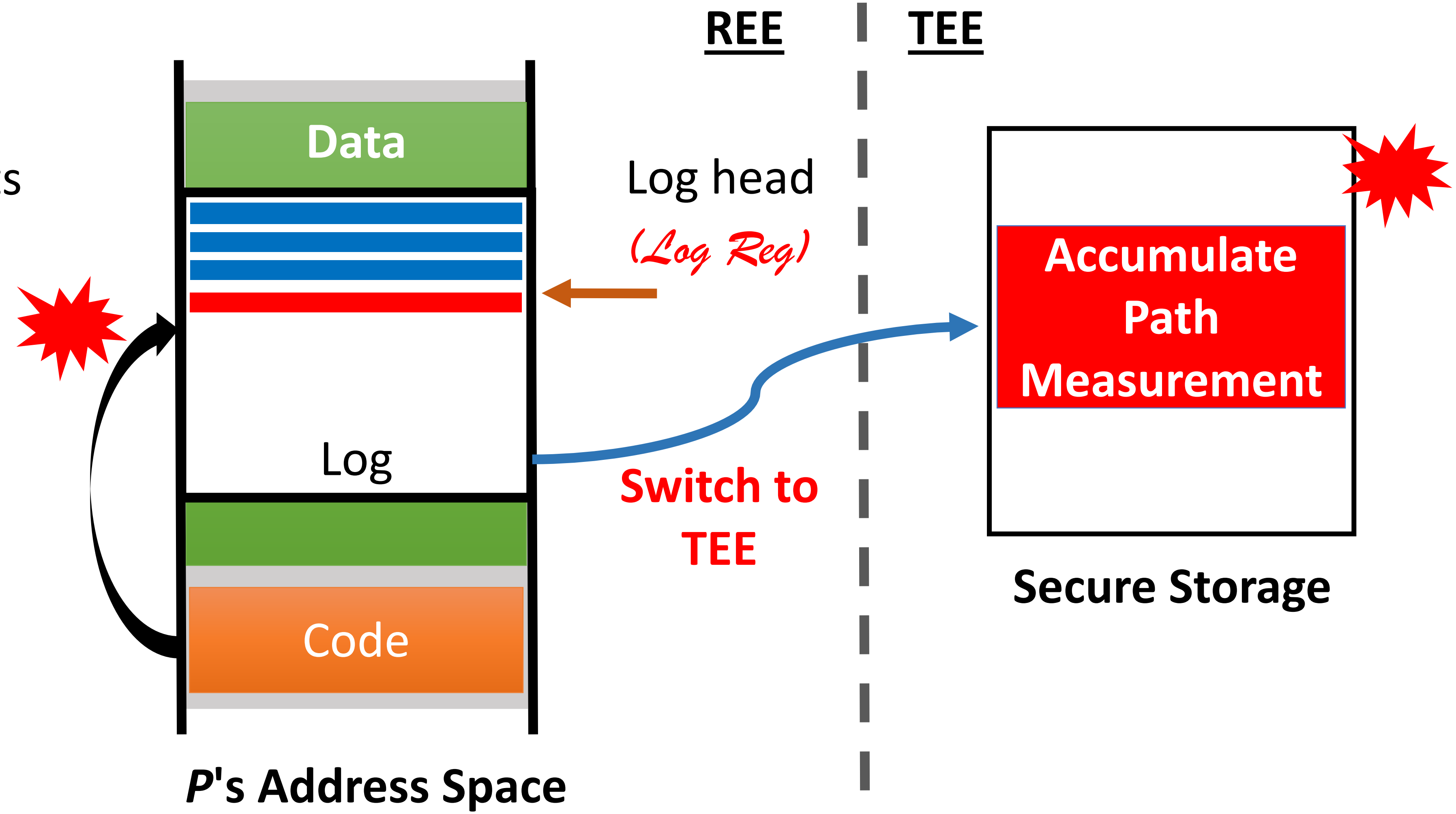


Buffer in Local Log

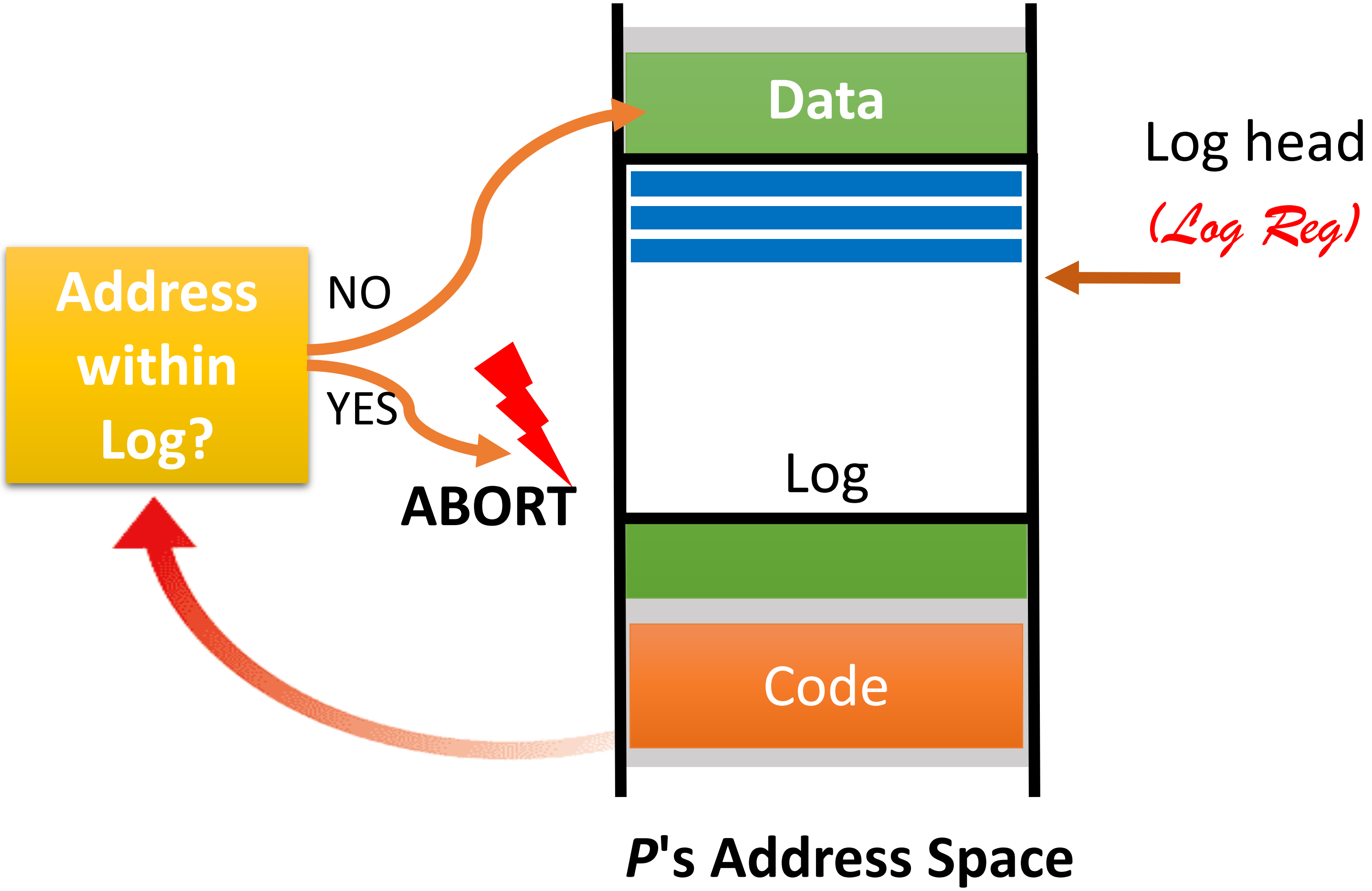


Corruption of Log Data

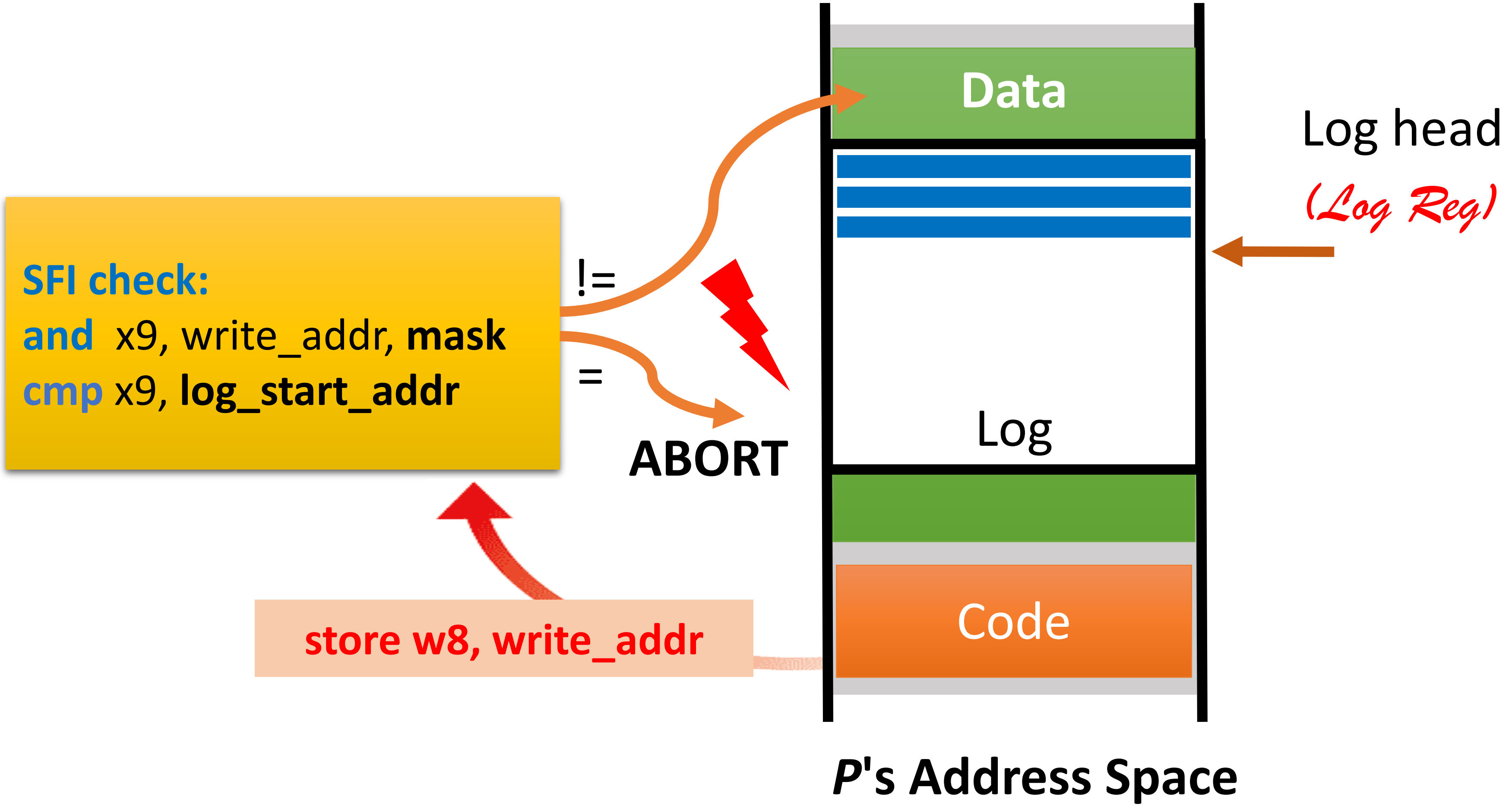
Problem:
P can write anywhere in its data region!



Protect the Log Data



Protect Log with Software Fault Isolation



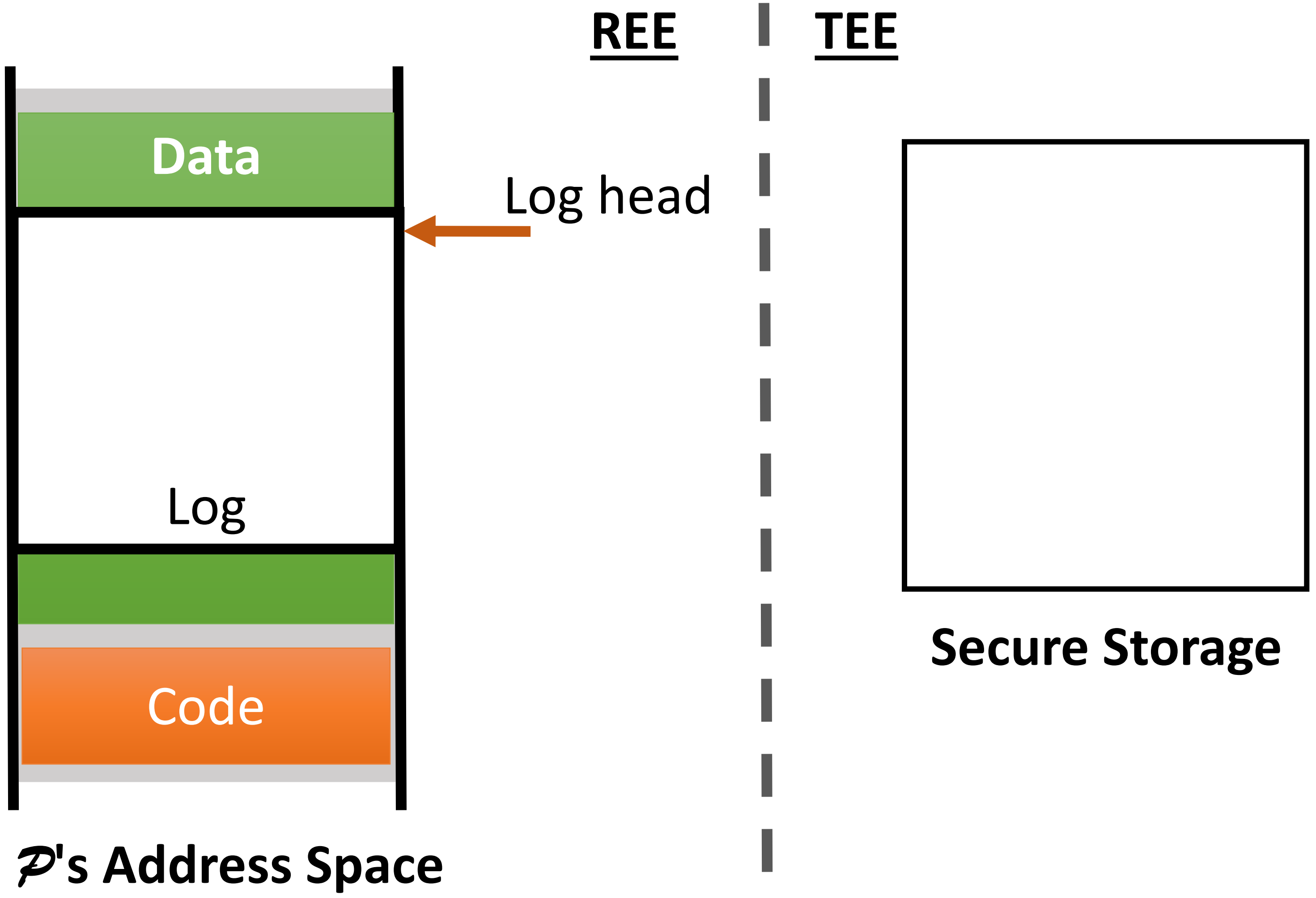
Key Contributions

1) Store path locally in log (reduces TEE domain switches)

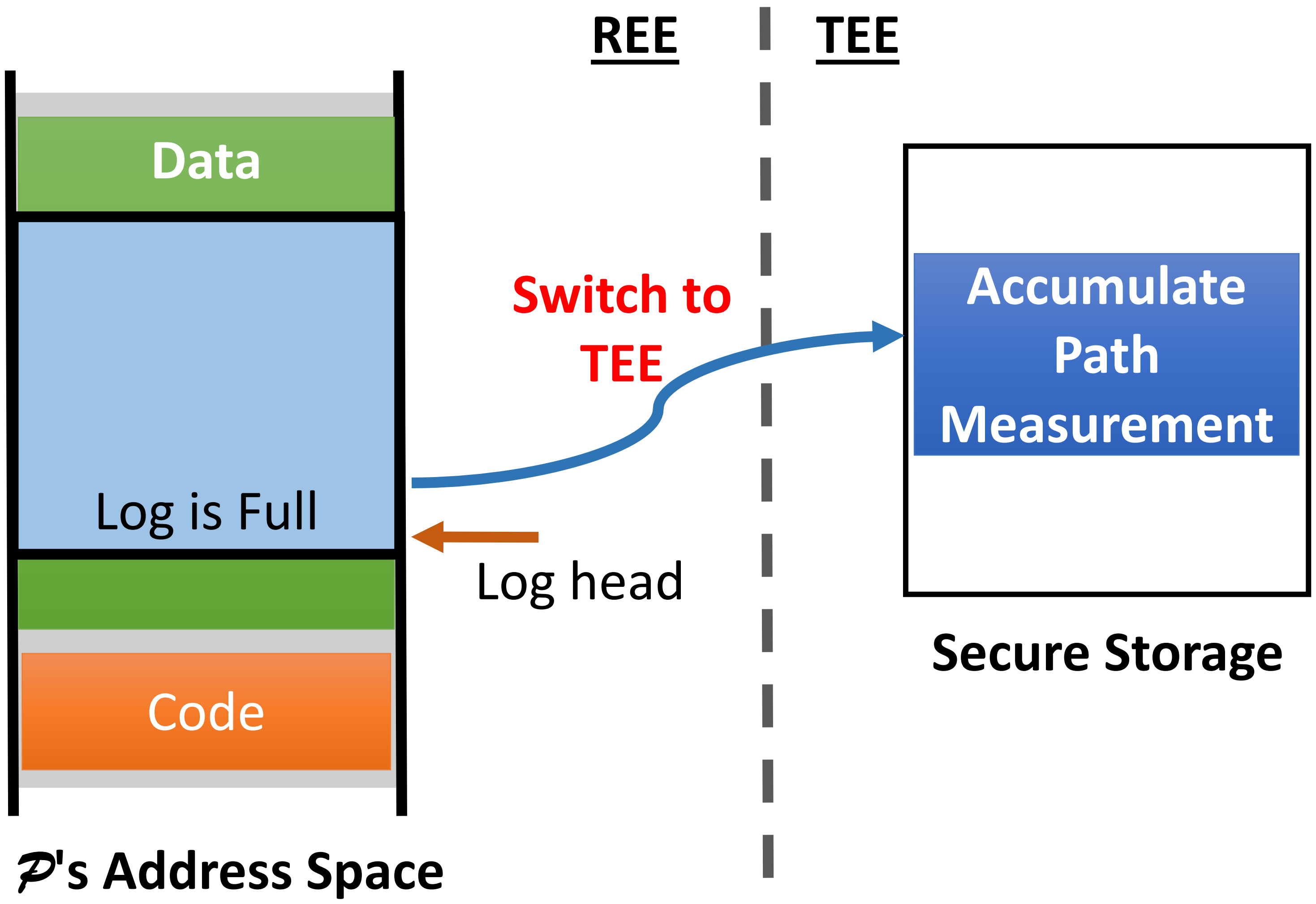
2) Instrument ***P*** using Ball Larus Profiling (reduces log entries)

3) Compact & expressive path representation

Flush Log to TEE



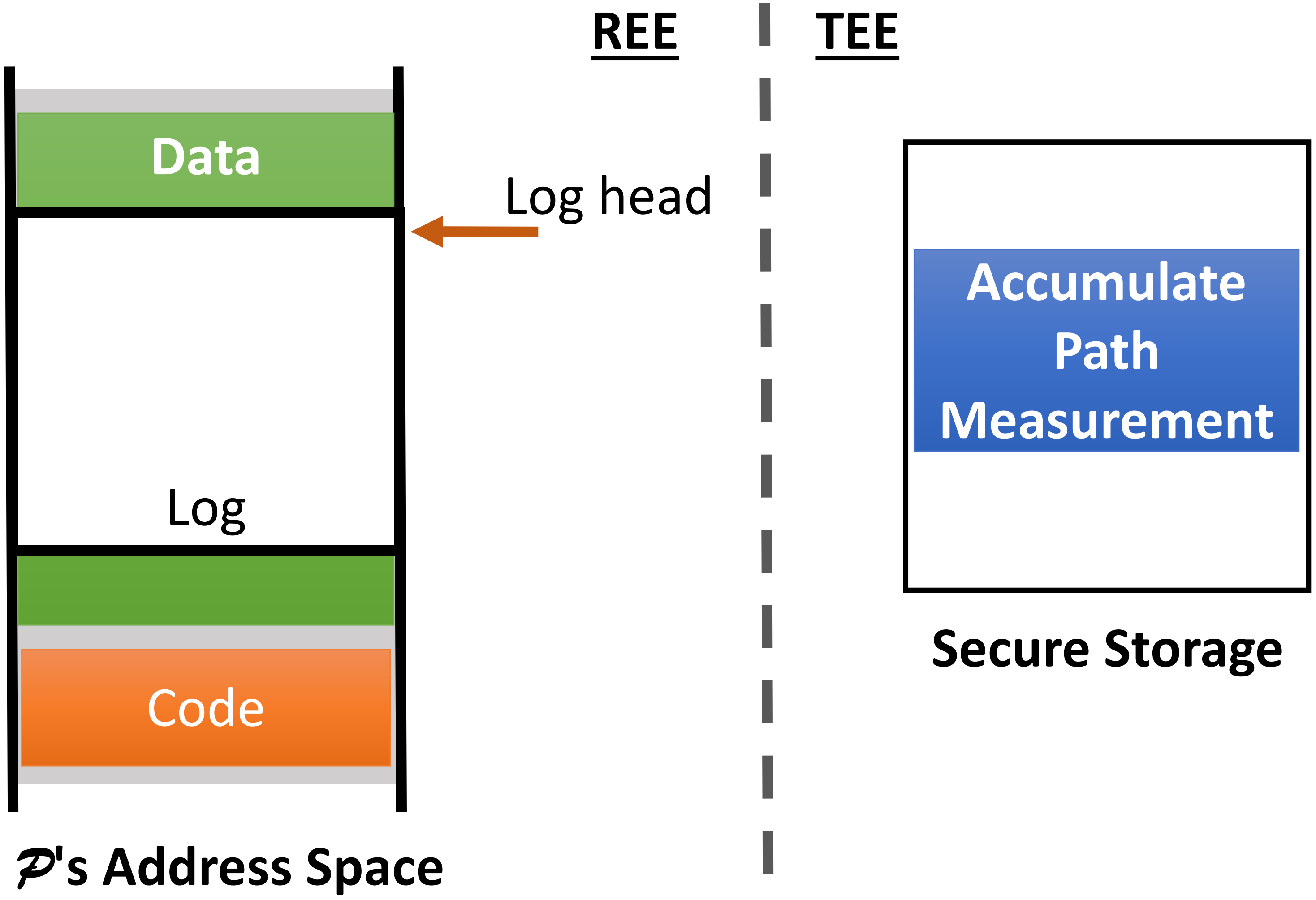
Flush Log to TEE



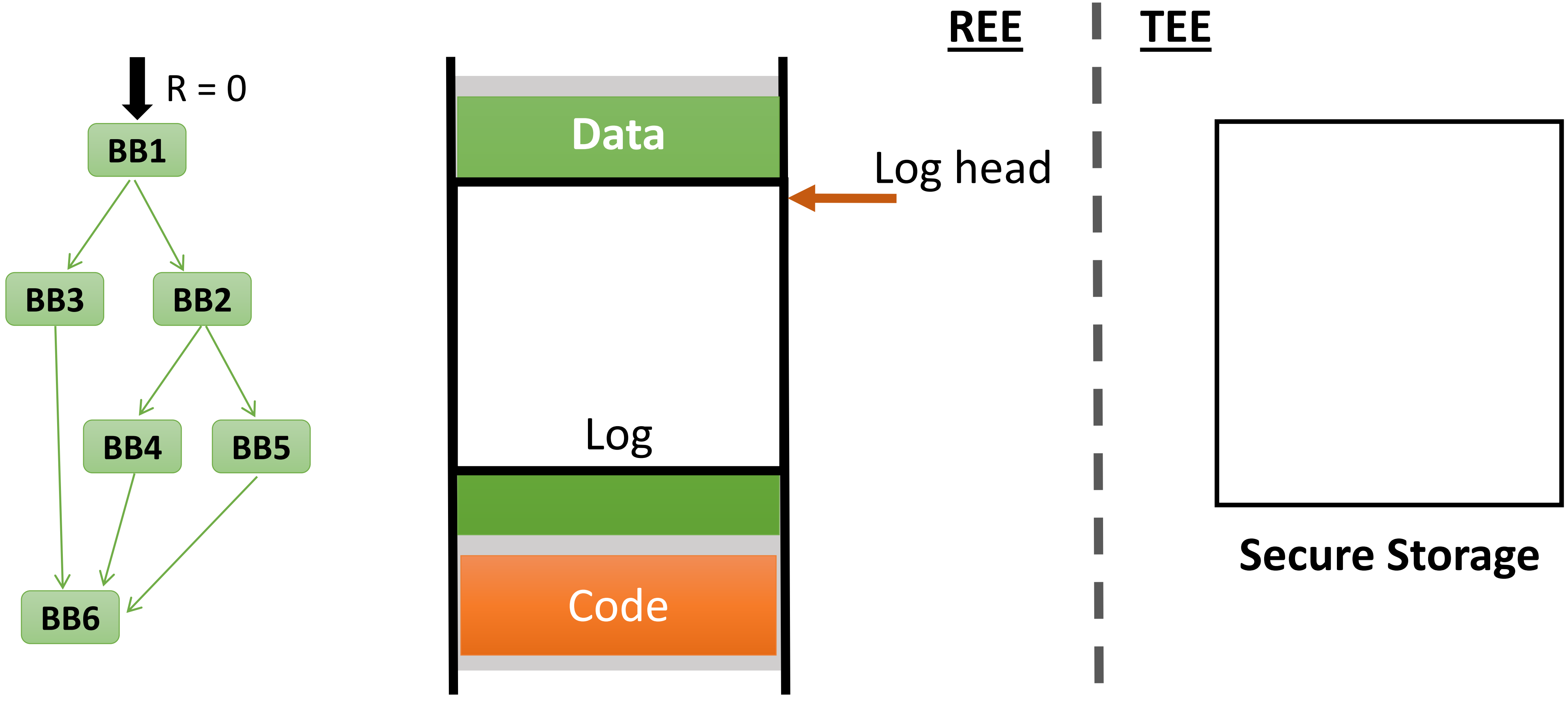
Flush Log to TEE

Number of TEE switches = Number of log flushes!

Problem: Reduce log flushes.



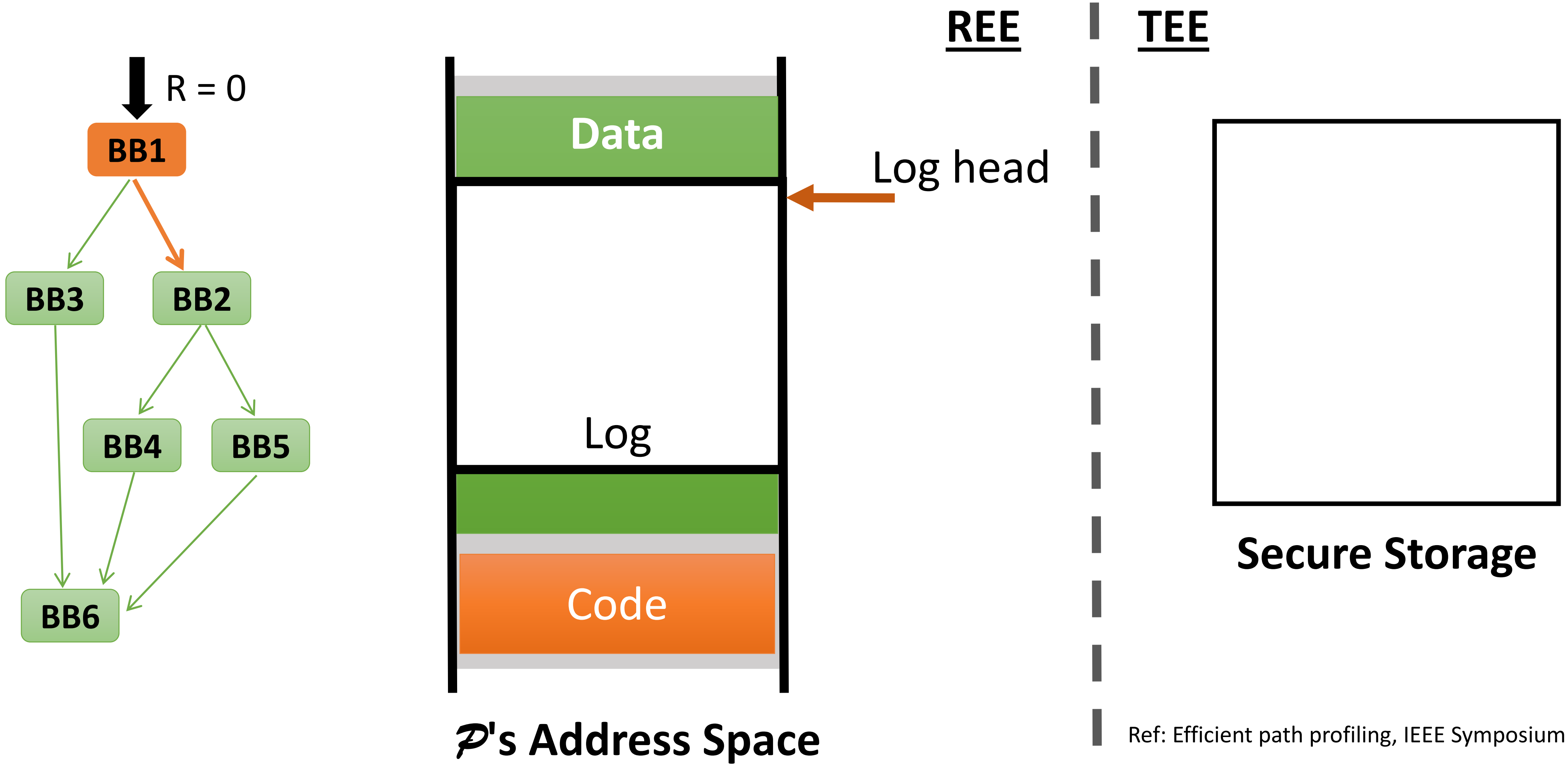
Reduce Log Entries



$\mathcal{P}$'s Address Space

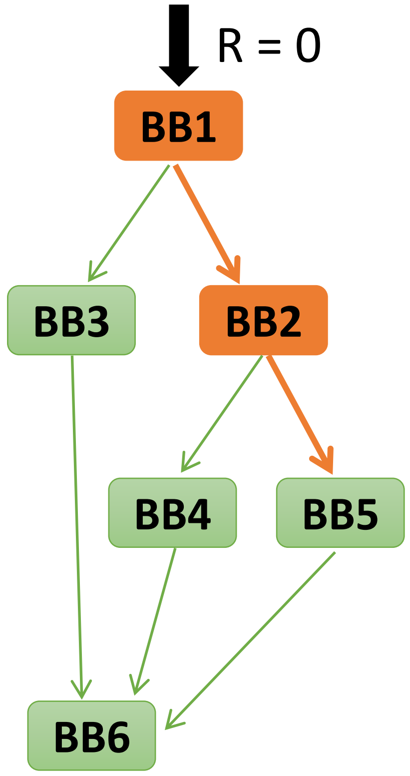
Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

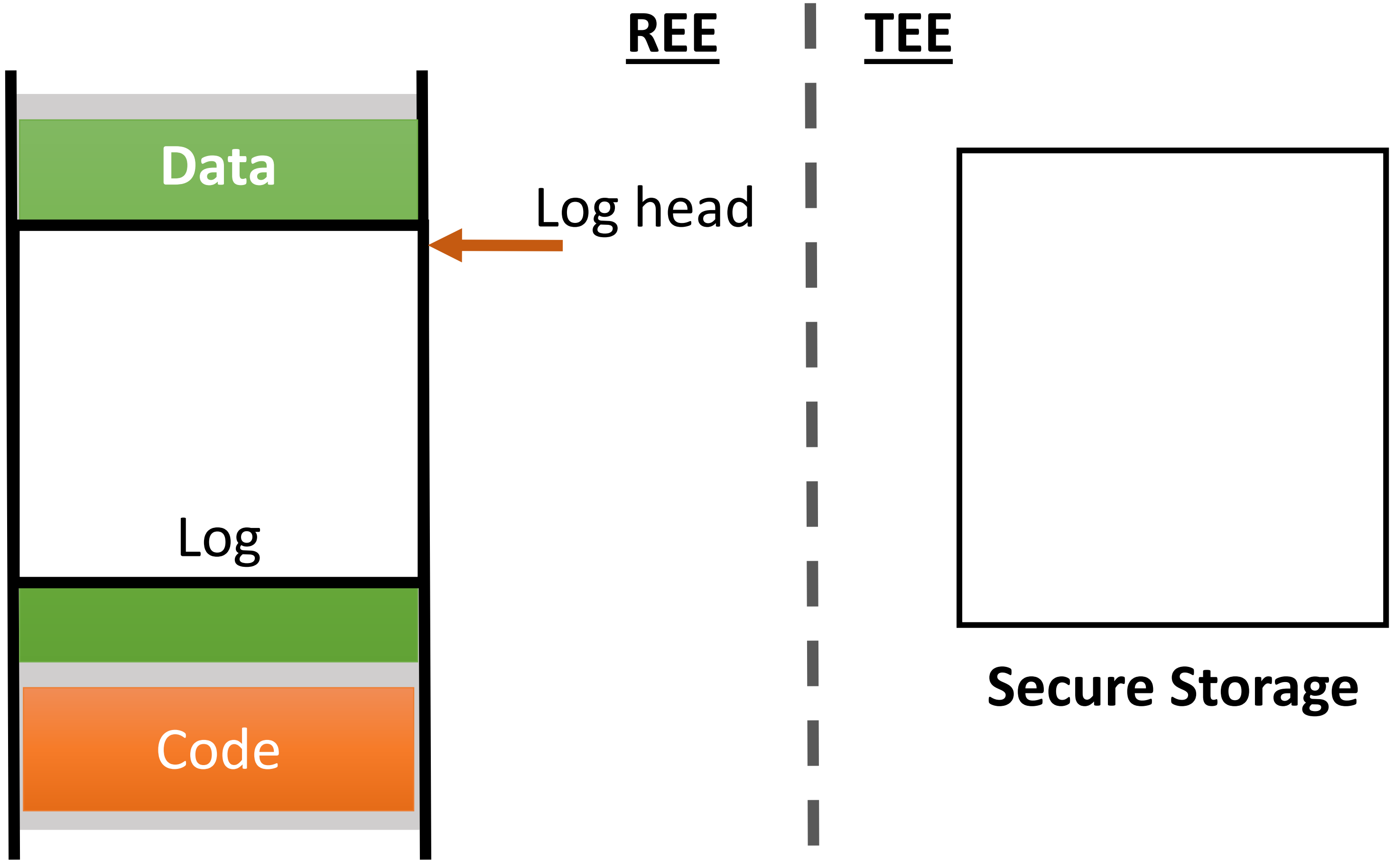
Reduce Log Entries



Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

Reduce Log Entries

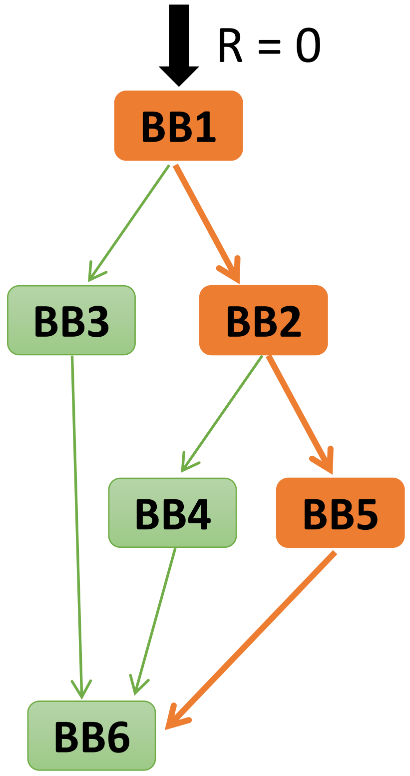


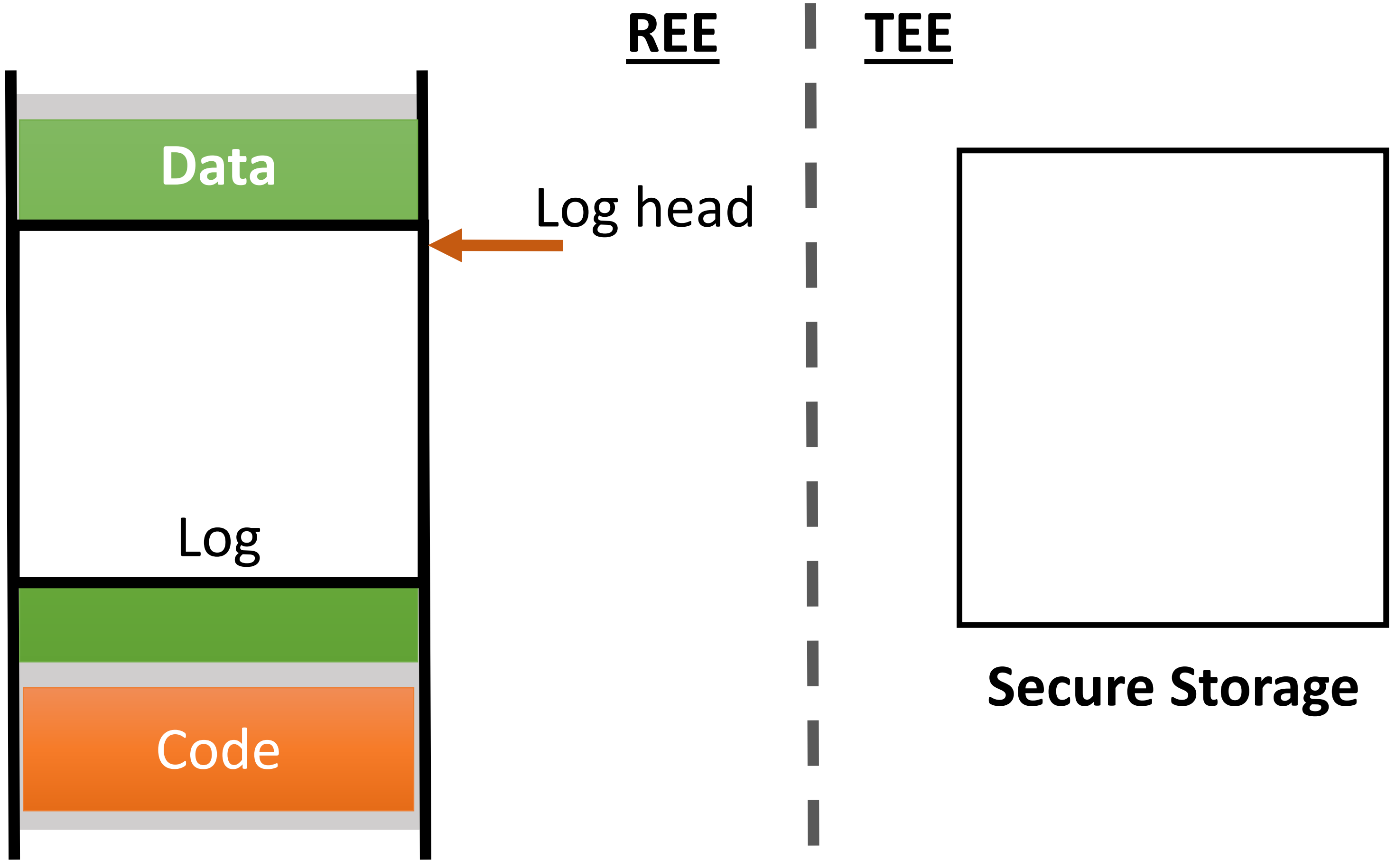


$\mathcal{P}$'s Address Space

Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

Reduce Log Entries

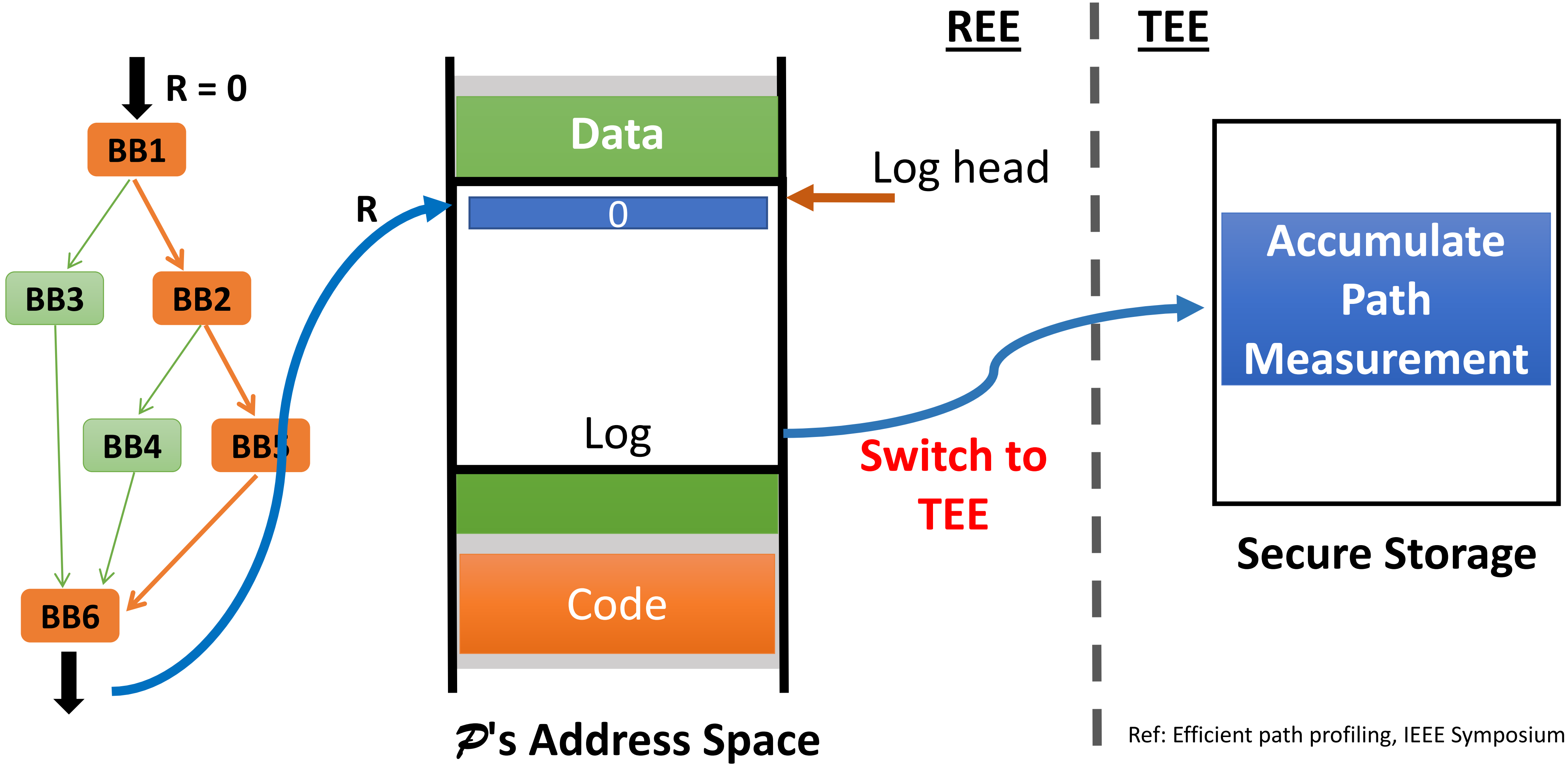




𝒫's Address Space

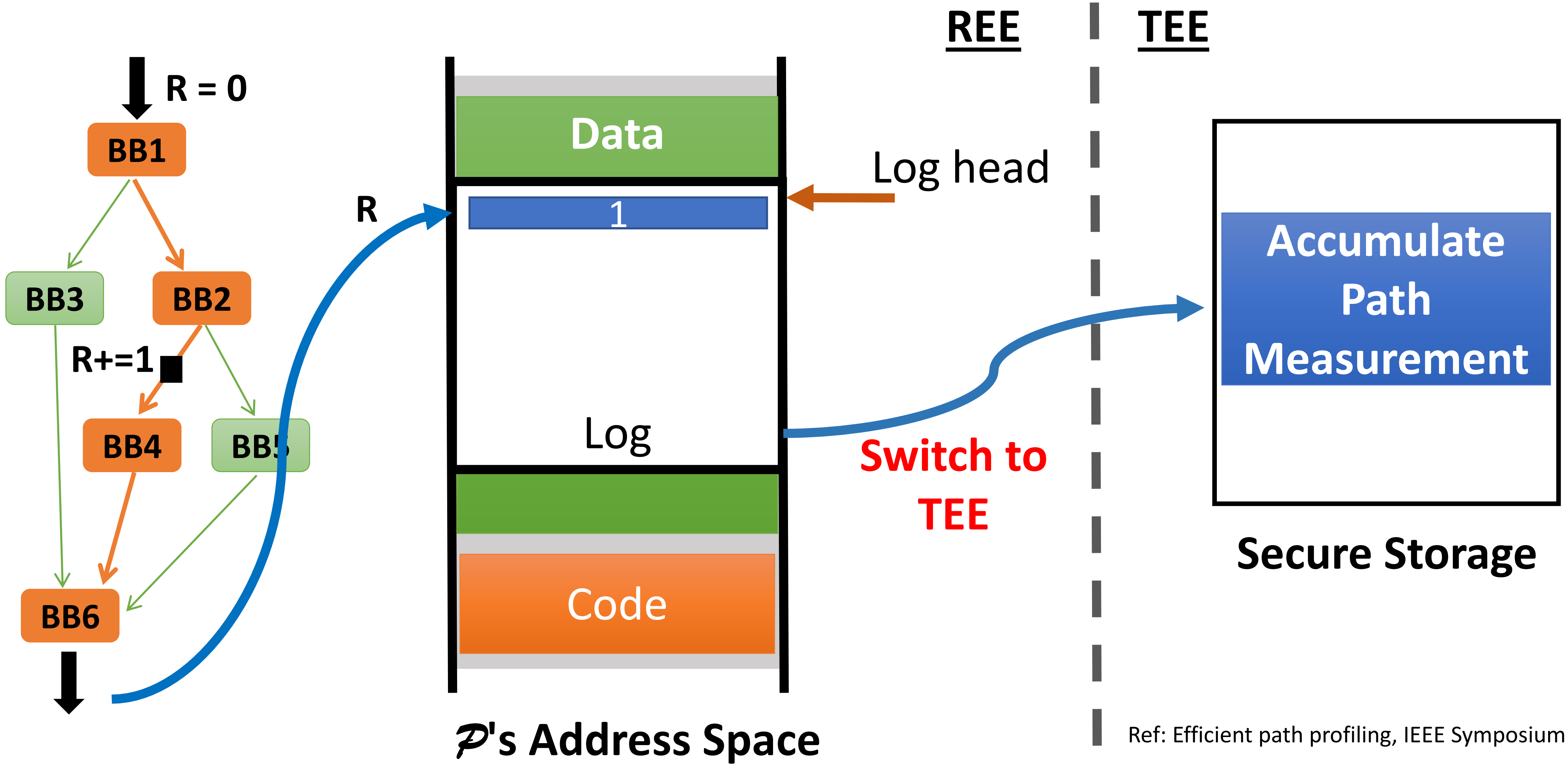
Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

Reduce Log Entries



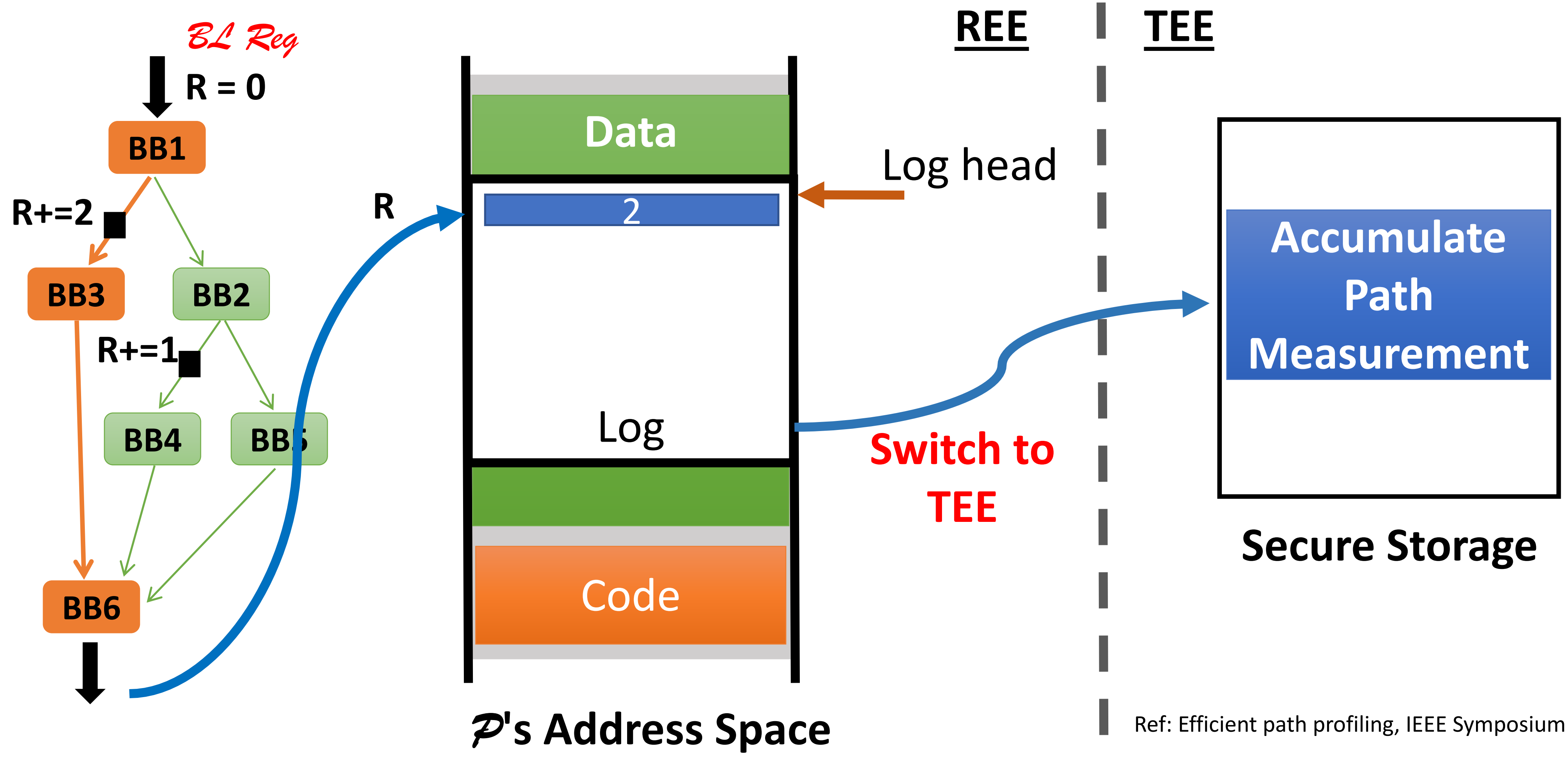
Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

Reduce Log Entries



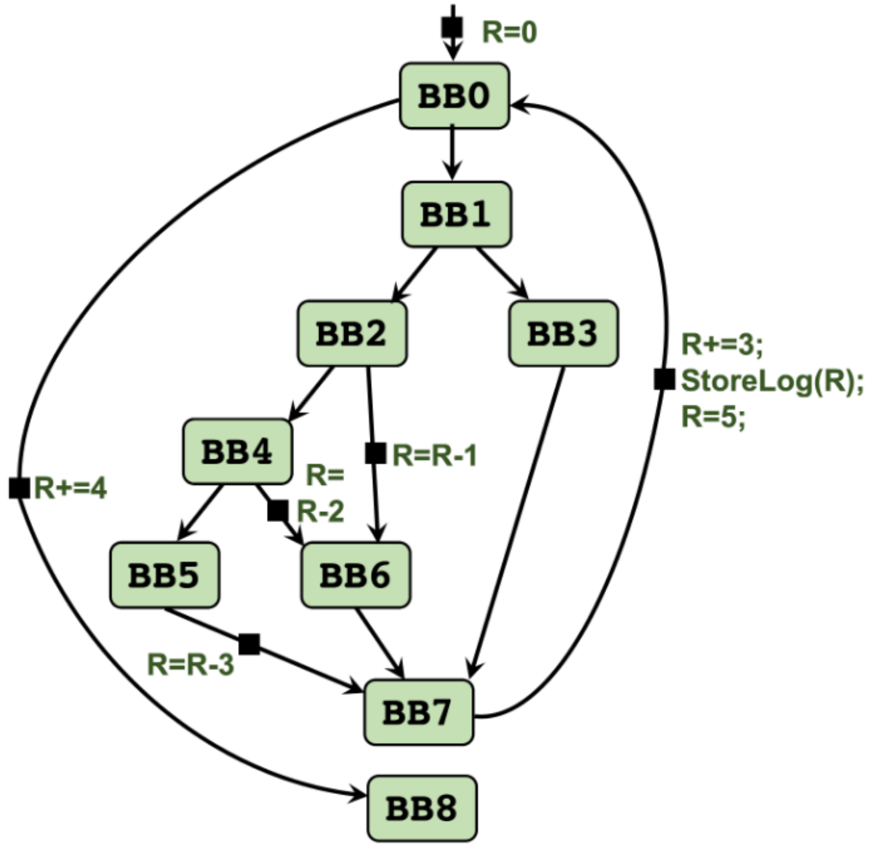
Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

Reduce Log Entries



Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

Ball Larus Profiling: Handling Loops



Path	Path ID
BB0->BB1->BB2->BB4->BB5->BB7	0
BB0->BB1->BB2->BB4->BB6->BB7	1
BB0->BB1->BB2->BB6->BB7	2
BB0->BB1->BB3->BB7	3
BB0->BB8	4
BB7->BB0->BB1->BB2->BB4->BB5->BB7	5
BB7->BB0->BB1->BB2->BB4->BB6->BB7	6
BB7->BB0->BB1->BB2->BB6->BB7	7
BB7->BB0->BB1->BB3->BB7	8
BB7->BB0->BB8	9

Ball Larus Instrumentation with Logging

We reserve physical register w20 for BL number *(BL Reg)* and physical register x19 for Log head *(Log Reg)*

Initialization on function entry:

```
mov w20, #0x0
```

Increment on edges:

```
add w20, w20, #increment_val
```

Loop header:

```
add w20, w20, #increment_val
str w20, [x19], #4
mov w20, #reset_val
```

Function call:

```
str w20, [x19], #4
mov w8, #func_entry_id
str w8, [x19], #4
bl func_addr <check_alarm>
mov w20, #reset_val
```

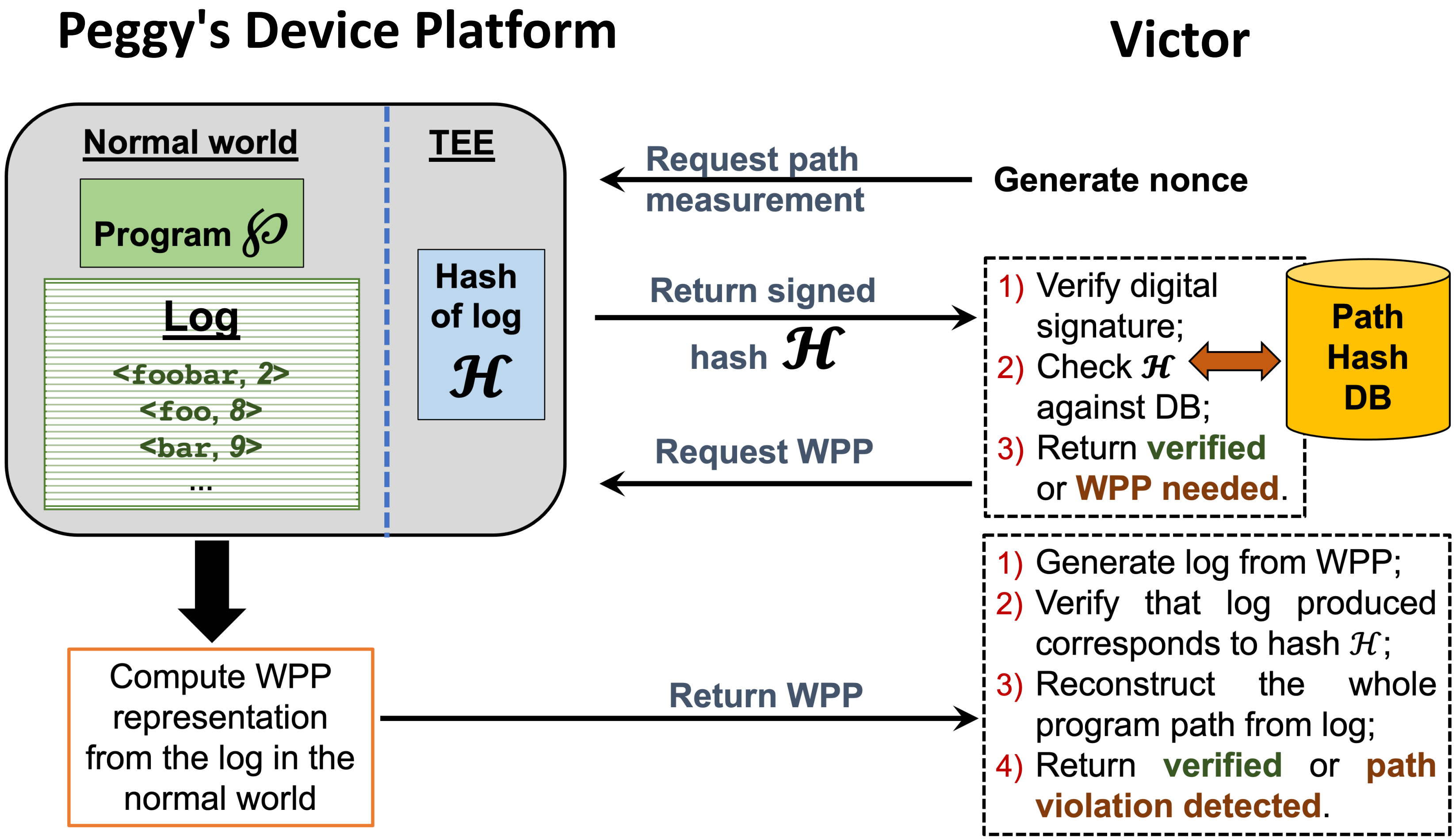
Function return/exit:

```
str w20, [x19], #4
mov w8, #func_exit_id
str w8, [x19], #4
str x30, [x19], #8
```

Reduction in Log entries using Ball Larus

Embench-IOT Program ↓	# Log entries using BLAST's approach	$\frac{\text{CFLAT}}{\text{BLAST}}$	$\frac{\text{OAT}}{\text{BLAST}}$
aha-mont64	206,847,012	**4.14×**	**1.90×**
crc32	523,090,012	**1.66×**	**0.66×**
cubic	710,012	**2.85×**	**1.21×**
edn	362,268,012	**3.95×**	**1.03×**
huffbench	235,422,012	**4.18×**	**2.11×**
malmult-int	387,552,454	**3.09×**	**1.05×**
minver	68,820,024	**4.03×**	**1.68×**
nbody	4,823,032	**3.58×**	**1.31×**
nettle-aes	52,884,268	**4.30×**	**1.49×**
nettle-sha256	31,825,020	**7.01×**	**1.07×**
primecount	282,283,012	**5.69×**	**3.18×**
sglib-combined	298,121,016	**4.90×**	**2.54×**
st	24,921,012	**1.74×**	**0.68×**
tarfind	121,062,486	**2.21×**	**0.97×**
ud	258,650,012	**2.21×**	**1.60×**

Workflow for Verification



WPP Representation

Repeated sequences of control-flow events are compressed into context-free grammar rules.

Log Entries	Identifier
<foobar, 2>	a
<foo, 8>	b
<bar, 9>	c
<foobar, 5>	d
<foo, 8>	b
<bar, 9>	c

Execution Trace: abcdbc

WPP:

S -> aCdC

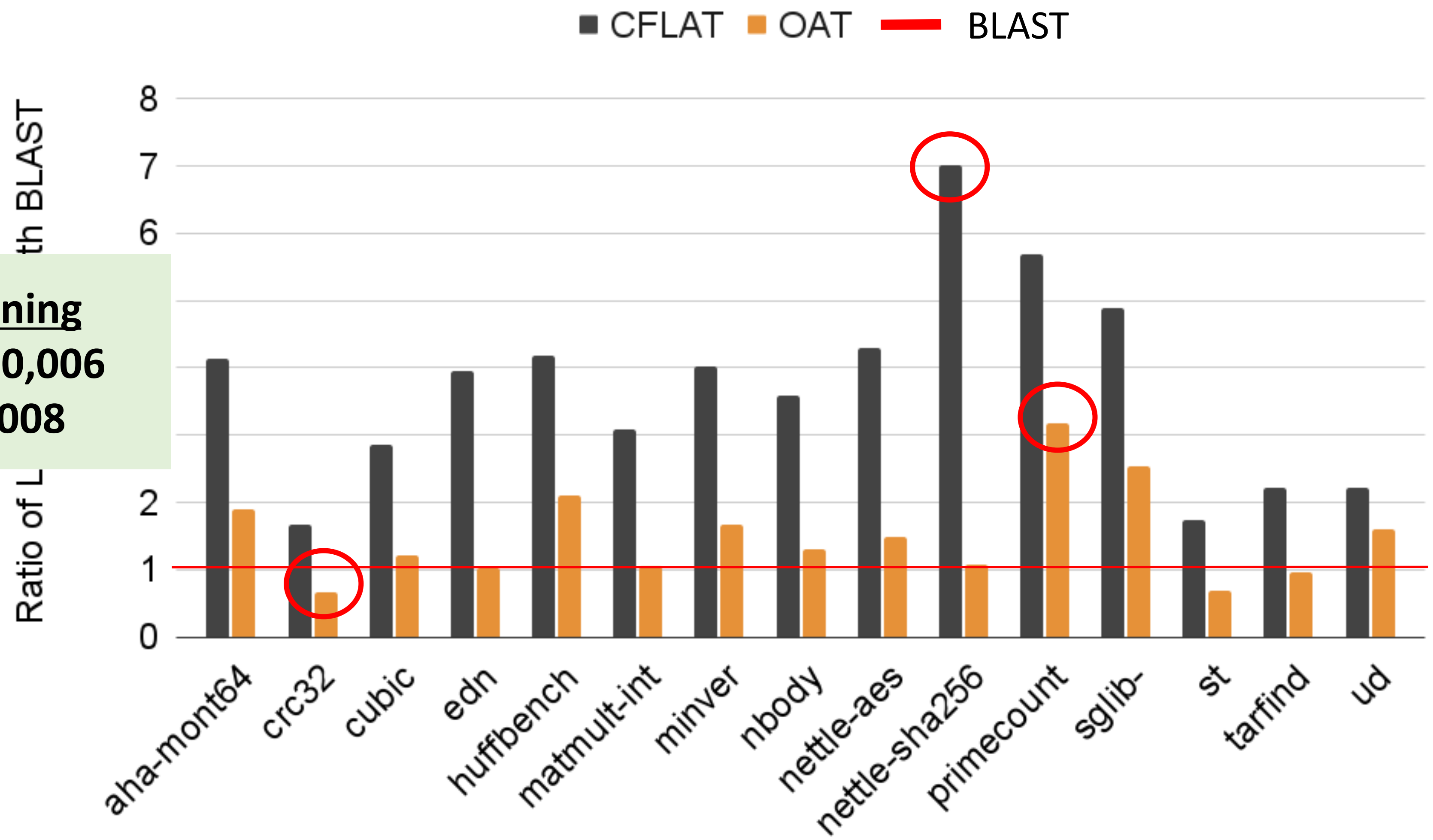
C -> bc

Ref: Whole program paths, ACM SIGPLAN Symposium on Programming Language Design and Implementation, 1999

Qualitative Security Analysis

1. Attacker modifies *BL Reg* suitably to record desired path value
   i. The *BL Reg* is reserved.
   ii. The indirect jump and call addresses are logged.
2. Attacker corrupts the Log
   i. Tries to use program's store instruction to write in Log
      - Prevented by SFI checks on all store instructions
   ii. Tries to use BLAST instrumentation to write in Log
      - The *Log Reg* is reserved, and it is only incremented by instrumentation.
      - It can only append to Log. **But the execution trace is always recorded!**

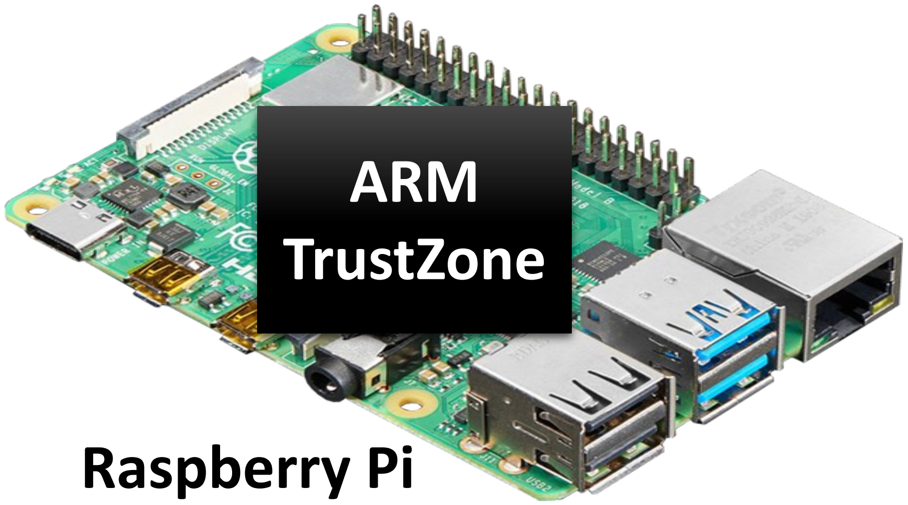
Effectiveness of Ball Larus Profiling



<u>crc32 with inlining</u>
BLAST: 348,670,006
OAT: 348,840,008

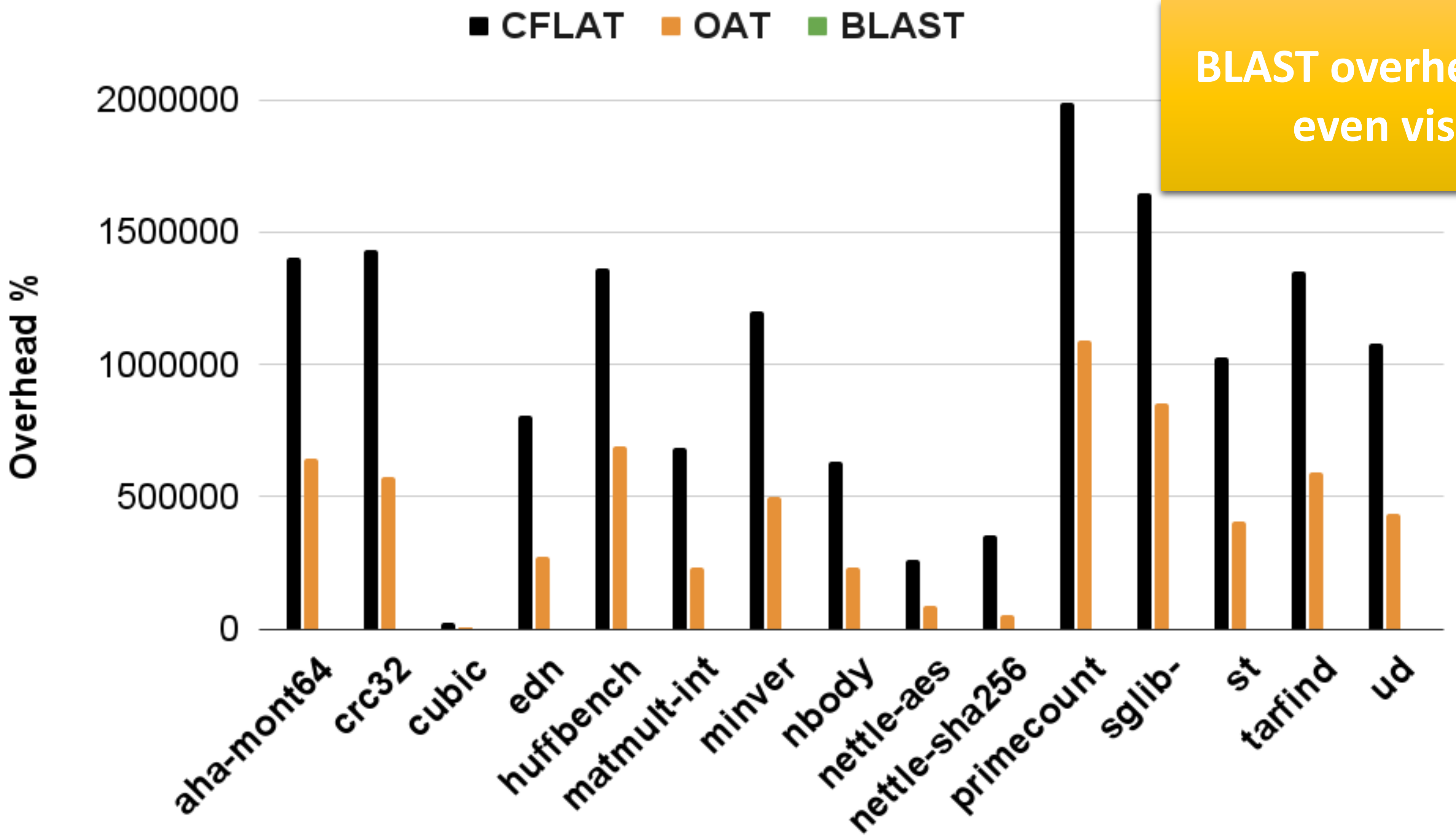
Experimental Setup

P → BLAST Compiler Pass → *P'* →



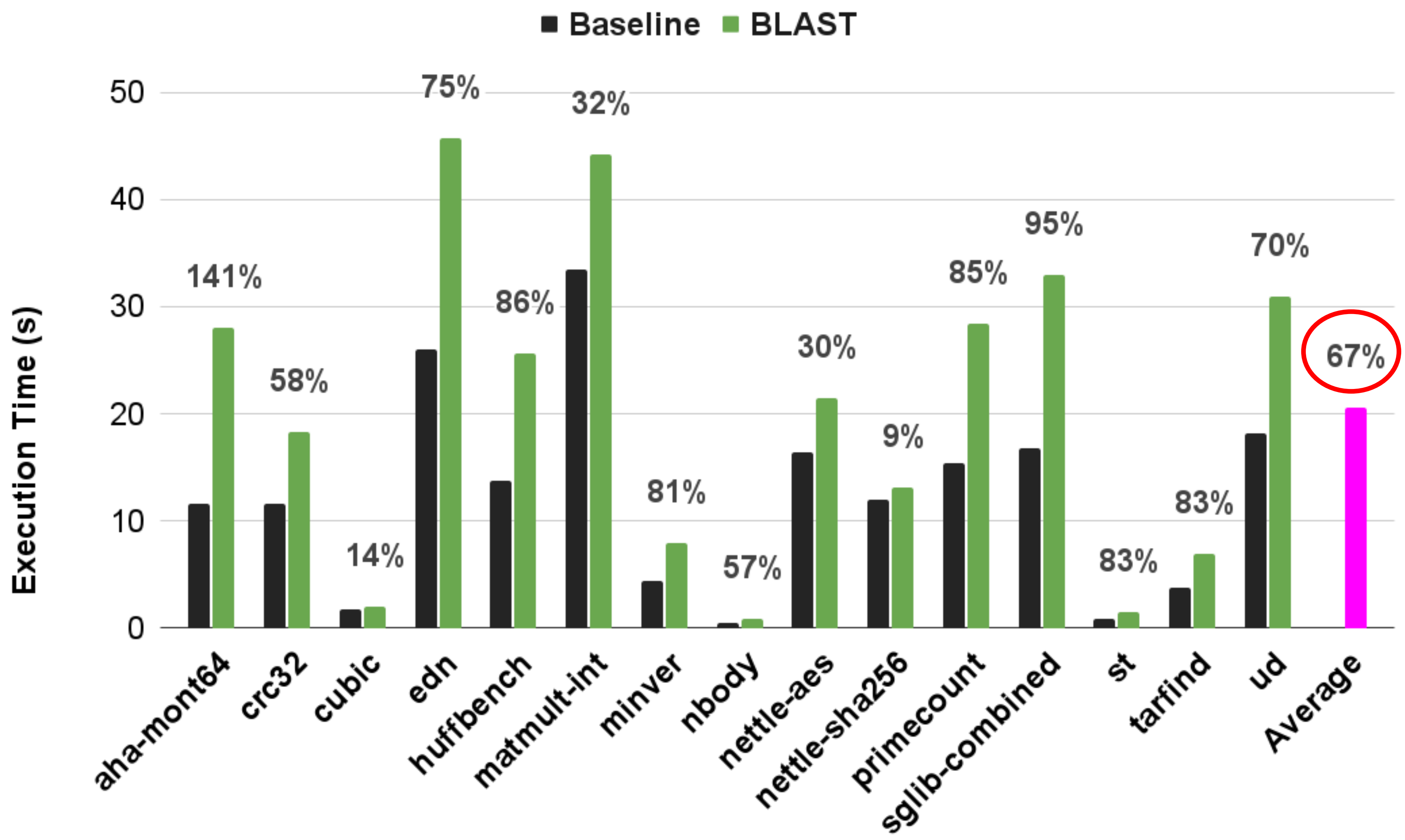
Benchmark: Embench-IoT (https://github.com/embench/embench-iot)

Comparison with CFLAT & OAT

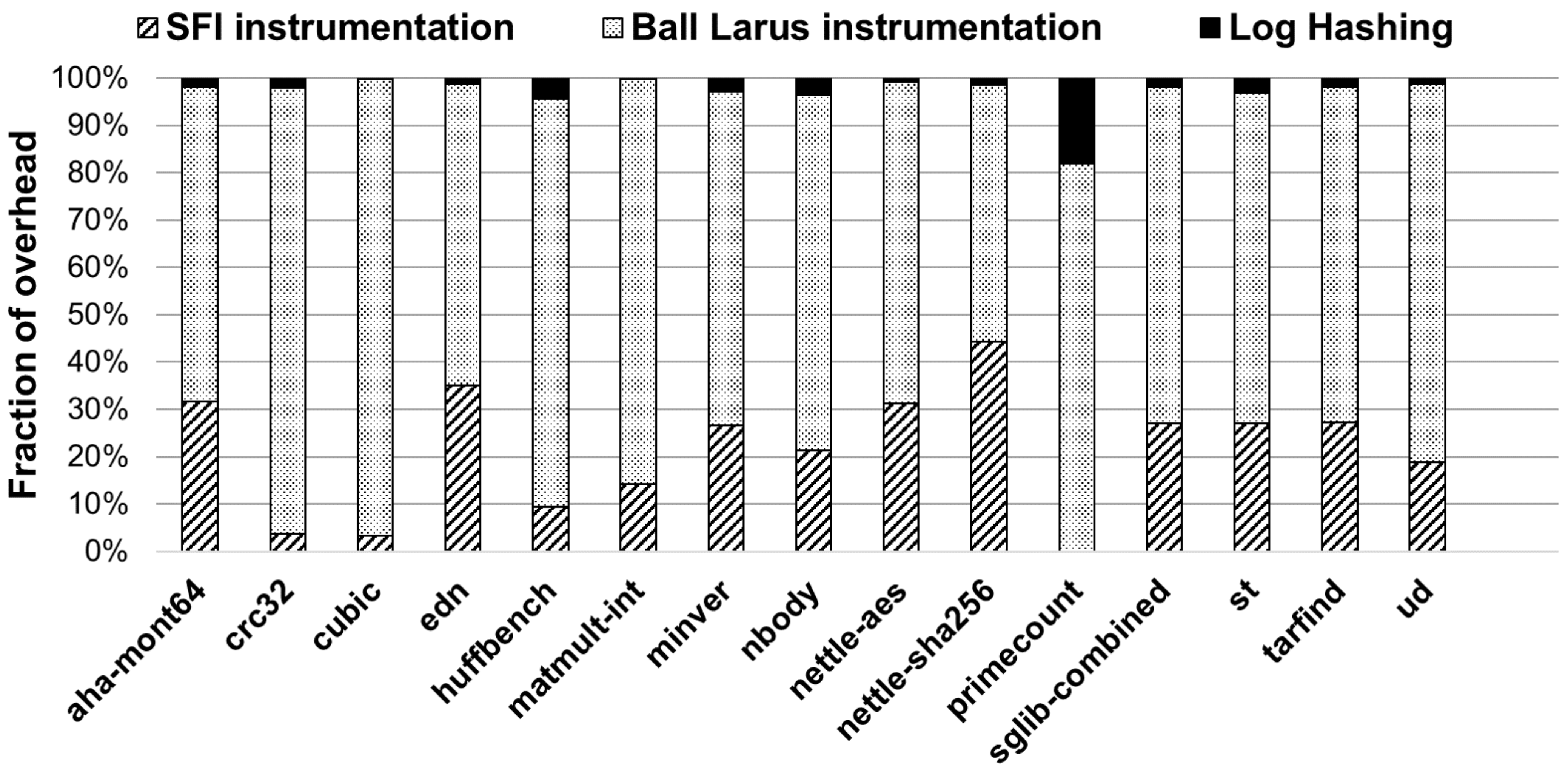


BLAST overhead is not even visible!

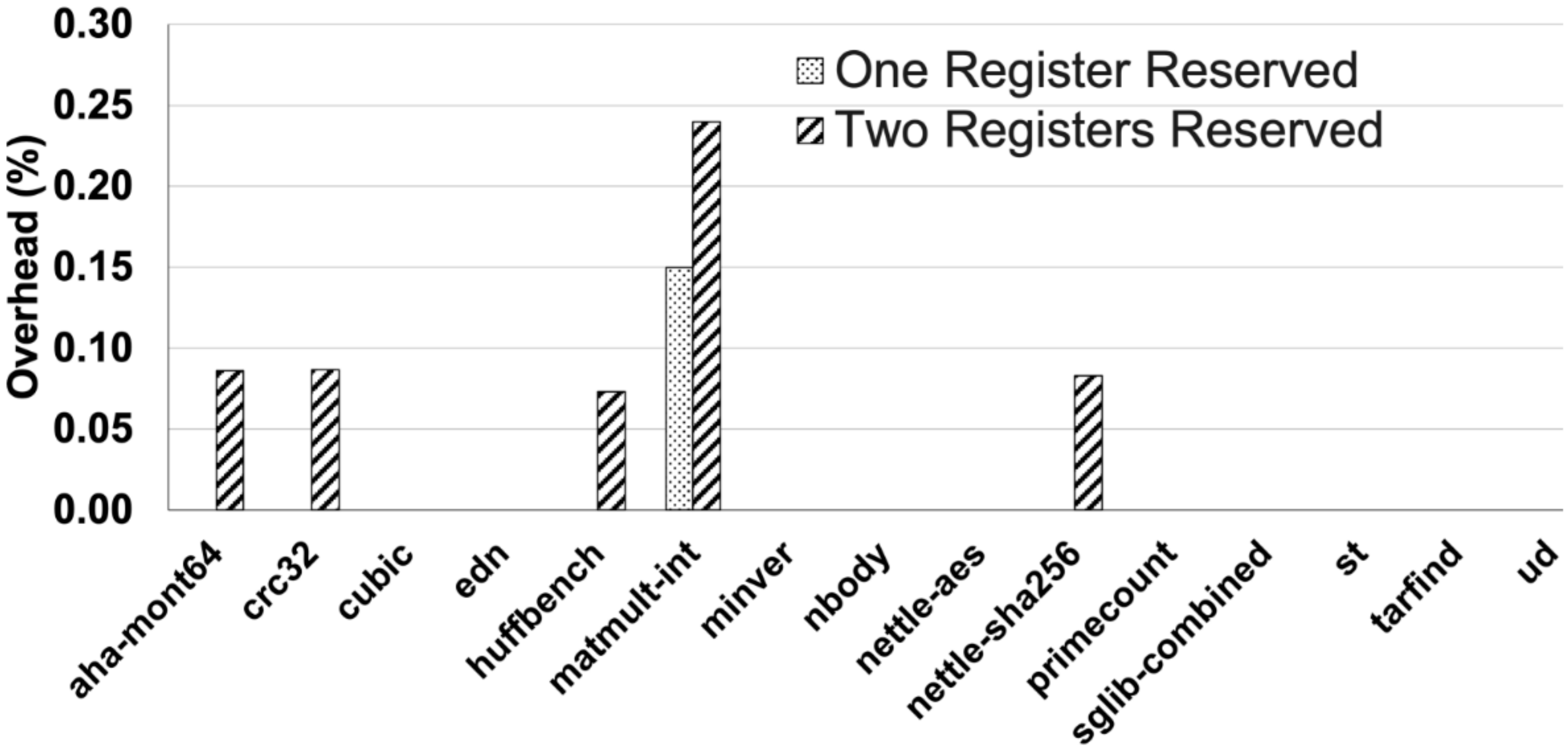
Performance of BLAST



Runtime Overhead Breakdown



Impact of Reserving Registers



Effectiveness of WPP Representation

Embench-IOT Program ↓	**Raw log size (MB)**	**bzip2 file size (bytes)**	**WPP size (bytes)**
aha-mont64	724.5MB	475,740 bytes	768 bytes
crc32	664.7MB	33,490 bytes	147 bytes
cubic	1.2MB	233 bytes	216 bytes
edn	1376.6MB	211,078 bytes	818 bytes
huffbench	889.8MB	4,706,860 bytes	9750 bytes
matmult-int	1477.7MB	105,882 bytes	370 bytes
minver	215.9MB	63,145 bytes	699 bytes
nbody	17.6MB	2,051 bytes	408 bytes
nettle-aes	195.2MB	40,022 bytes	843 bytes
nettle-sha256	132.3MB	35,055 bytes	336 bytes
primecount	1076.8MB	23,034,525 bytes	73,478 bytes
sglib-combined	910.0MB	421,6020 bytes	6,716 bytes
st	34.7MB	3,784 bytes	476 bytes
tarfind	184.6MB	382,229 bytes	257,756 bytes
ud	975.4MB	297,473 bytes	533 bytes

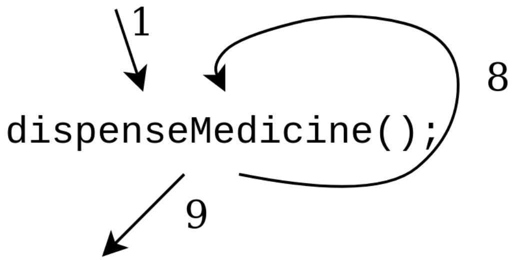
Case Study - Syringe Pump

Open Syringe Pump Code

Code

```
for (i=0; i<steps; i++)
    dispenseMedicine();
```

Paths



WPPs

Bolus = 0.010 ml	**Bolus = 0.011 ml**
Execution path trace:	Execution path trace:
1 8 *(repeated **67 times**)* 9	1 8 *(repeated **74 times**)* 9
S -> 1 AA**EF** 9	S -> 1 AA**CE** 9
A -> BB	A -> BB
B -> CC	B -> CC
C -> DD	C -> DD
D -> EE	D -> EE
E -> FF	E -> FF
F -> 8	F -> 8

Syringe Pump Benchmark

Bolus (mL)	Baseline Time(s)	BLAST Time(s)	BLAST Raw Overhead (s)	CFLAT Raw Overhead (s)
0.5 mL	1.28	1.42	**0.14 (10%)**	**1.2 (93%)**
1 mL	2.56	2.71	**0.15 (5%)**	**2.4 (93%)**
2 mL	5.12	5.28	**0.16 (3%)**	**4.8 (93%)**

ACM Conference on Computer and Communications Security (CCS) 2023

Whole-Program Control-flow Path Attestation

Nikita Yadav and Vinod Ganapathy

Computer Systems Security Laboratory
Indian Institute of Science, Bangalore

nikitayadav@iisc.ac.in, vg@iisc.ac.in