Whole-Program Control-flow Path Attestation

Nikita Yadav and Vinod Ganapathy

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

Victor (Verifier)  
Peter (Prover)  

Command  
Path DB  
Path commitment  
*e.g., set of executed instructions*  
Verify  
Secure Hardware  
Record the path  

Verified or path violation detected
Background & Threat Model

Peter's Device

Normal World/REE

- User mode
- Privileged mode

Secure World/TEE

- User mode
- Privileged mode
- Secure Monitor

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**

<table>
<thead>
<tr>
<th>Normal World/REE</th>
<th>Secure World/TEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>User mode</td>
</tr>
<tr>
<td>User mode</td>
<td>User mode</td>
</tr>
<tr>
<td>Privileged mode</td>
<td>Privileged mode</td>
</tr>
<tr>
<td></td>
<td>Secure Monitor</td>
</tr>
</tbody>
</table>

**Possible Threats:**

1. $P$ could be modified
2. Code injection in $P$
4. Input corruption/Data corruption
5. Out of scope – Physical attacks.

---

**Raspberry Pi**

 ARM TrustZone

Whole-program Control-flow Path Attestation
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
Possible Threats:

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

Record program execution path securely.
Strawman Approach I

Whole-program Control-flow Path Attestation
Strawman Approach I

Switch to TEE

Whole-program Control-flow Path Attestation
Strawman Approach I

Whole-program Control-flow Path Attestation
Strawman Approach

Switch to TEE

Whole-program Control-flow Path Attestation
Strawman Approach II

Whole-program Control-flow Path Attestation
Strawman Approach II

Whole-program Control-flow Path Attestation
Strawman Approach II

Whole-program Control-flow Path Attestation
Strawman Approach II

Whole-program Control-flow Path Attestation
Strawman Approach II

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

Whole-program Control-flow Path Attestation
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

<table>
<thead>
<tr>
<th>Embench-IoT Benchmark</th>
<th>Strawman Approach I (CFLAT)</th>
<th>Strawman Approach II (OAT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aha-mont64</td>
<td>857,844,016</td>
<td>392,967,008</td>
</tr>
<tr>
<td>crc32</td>
<td>871,930,016</td>
<td>348,840,008</td>
</tr>
<tr>
<td>cubic</td>
<td>2,030,022</td>
<td>860,013</td>
</tr>
<tr>
<td>edn</td>
<td>1,106,118,020</td>
<td>372,621,011</td>
</tr>
<tr>
<td>huffbench</td>
<td>984,236,016</td>
<td>496,903,008</td>
</tr>
<tr>
<td>matmul-int</td>
<td>1,201,018,222</td>
<td>406,825,691</td>
</tr>
<tr>
<td>minver</td>
<td>277,500,079</td>
<td>115,440,042</td>
</tr>
<tr>
<td>nb body</td>
<td>17,279,126</td>
<td>6,329,070</td>
</tr>
<tr>
<td>nettle-aes</td>
<td>227,449,298</td>
<td>78,858,777</td>
</tr>
<tr>
<td>nettle-sha256</td>
<td>223,250,050</td>
<td>34,200,025</td>
</tr>
<tr>
<td>primecount</td>
<td>1,607,180,016</td>
<td>880,206,008</td>
</tr>
</tbody>
</table>
Effect of TEE switches on Runtime

<table>
<thead>
<tr>
<th>Embench-IoT Benchmark</th>
<th>Total TEE domain Switches Encountered at Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CFLAT</td>
</tr>
<tr>
<td>nettle-sha256</td>
<td>223,250,050</td>
</tr>
</tbody>
</table>

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

Baseline Execution Time

<table>
<thead>
<tr>
<th>Baseline Execution Time</th>
<th>Time with CFLAT</th>
<th>Time with OAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 seconds</td>
<td>&gt; 11 hours</td>
<td>~ 2 hours</td>
</tr>
</tbody>
</table>

CFLAT and OAT impose over 1000× 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( ) {
    ........
    scanf("%d", &n) ← input: 5
    ........
    if (flag>0) func2( );
    else func3(n);
}

void func2( ) {
    ....
}

void func3(int n ) {
    n = 3
    ........
    while (n) {
        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

Whole-program Control-flow Path Attestation
Buffer in Local Log

REE

Secure Storage

TEE

Log head

P's Address Space

Data

Code

Log head

BB1

BB3

BB2

BB4

BB5

BB6
Buffer in Local Log

Whole-program Control-flow Path Attestation

REE

TEE

Secure Storage

P's Address Space

Data

Log head

Log

Code

BB1
BB2
BB3
BB4
BB5
BB6
Buffer in Local Log

Whole-program Control-flow Path Attestation
Buffer in Local Log

Whole-program Control-flow Path Attestation
Corruption of Log Data

Problem:
P can write anywhere in its data region!
Protect the Log Data

Whole-program Control-flow Path Attestation

Address within Log?

NO

YES

Log head
(Log Reg)

ABORT

Data

Log

Code

P's Address Space
Protect Log with Software Fault Isolation

SFI check:
\[
\text{and} \quad x9, \text{write_addr}, \text{mask}
\]
\[
\text{cmp} \quad x9, \text{log_start_addr}
\]

If \( \neq \), ABORT

store w8, write_addr

Log head
(Log Reg)

P's Address Space
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

REE

Data

Log head

TEL

Secure Storage

'P's Address Space

Whole-program Control-flow Path Attestation
Flush Log to TEE

REE

Secure Storage

Log is Full

Switch to TEE

Log head

 Accumulate Path Measurement

TEE

Data

Code

\(\mathcal{P}'s\ \text{Address Space}\)
Flush Log to TEE

Number of TEE switches = Number of log flushes!

Problem: Reduce log flushes.

REE

Data

Log head

Log

Code

\( \mathcal{P}'s \ Address \ Space \)

TEE

Accumulate Path Measurement

Secure Storage

Whole-program Control-flow Path Attestation
Reduce Log Entries

Whole-program Control-flow Path Attestation

Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996
Reduce Log Entries

Whole-program Control-flow Path Attestation

Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996
Reduce Log Entries

Whole-program Control-flow Path Attestation

Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996
Reduce Log Entries

Efficient path profiling, IEEE Symposium on Microarchitecture, 1996

Whole-program Control-flow Path Attestation
Reduce Log Entries

Whole-program Control-flow Path Attestation

Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996
Reduce Log Entries

Whole-program Control-flow Path Attestation

Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996
Reduce Log Entries

Whole-program Control-flow Path Attestation

Ref: Efficient path profiling, IEEE Symposium on Microarchitecture, 1996
Ball Larus Profiling: Handling Loops

<table>
<thead>
<tr>
<th>Path</th>
<th>Path ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB0-&gt;BB1-&gt;BB2-&gt;BB4-&gt;BB5-&gt;BB7</td>
<td>0</td>
</tr>
<tr>
<td>BB0-&gt;BB1-&gt;BB2-&gt;BB4-&gt;BB6-&gt;BB7</td>
<td>1</td>
</tr>
<tr>
<td>BB0-&gt;BB1-&gt;BB2-&gt;BB6-&gt;BB7</td>
<td>2</td>
</tr>
<tr>
<td>BB0-&gt;BB1-&gt;BB3-&gt;BB7</td>
<td>3</td>
</tr>
<tr>
<td>BB0-&gt;BB8</td>
<td>4</td>
</tr>
<tr>
<td>BB7-&gt;BB0-&gt;BB1-&gt;BB2-&gt;BB4-&gt;BB5-&gt;BB7</td>
<td>5</td>
</tr>
<tr>
<td>BB7-&gt;BB0-&gt;BB1-&gt;BB2-&gt;BB4-&gt;BB6-&gt;BB7</td>
<td>6</td>
</tr>
<tr>
<td>BB7-&gt;BB0-&gt;BB1-&gt;BB2-&gt;BB6-&gt;BB7</td>
<td>7</td>
</tr>
<tr>
<td>BB7-&gt;BB0-&gt;BB1-&gt;BB3-&gt;BB7</td>
<td>8</td>
</tr>
<tr>
<td>BB7-&gt;BB0-&gt;BB8</td>
<td>9</td>
</tr>
</tbody>
</table>

Whole-Program Control-Flow Path Attestation
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:**

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

---

**Increment on edges:**

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

---

**Loop header:**

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

---

**Function call:**

```asm
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:**

```asm
str w20, [x19], #4
mov w8, #func_exit_id
str w8, [x19], #4
str x30, [x19], #8
```
# Reduction in Log entries using Ball Larus

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

Peggy's Device Platform

Normal world
Program $\Phi$

Log
$<\text{foobar, 2}>$
$<\text{foo, 8}>$
$<\text{bar, 9}>$
...

Hash of log $\mathcal{H}$

Compute WPP representation from the log in the normal world

Request path measurement

Return signed hash $\mathcal{H}$

Request WPP

Return WPP

Victor

Generate nonce

1) Verify digital signature;
2) Check $\mathcal{H}$ against DB;
3) Return verified or WPP needed

Path Hash DB

1) Generate log from WPP;
2) Verify that log produced corresponds to hash $\mathcal{H}$;
3) Reconstruct the whole program path from log;
4) Return verified or path violation detected.
Repeated sequences of control-flow events are compressed into context-free grammar rules.

<table>
<thead>
<tr>
<th>Log Entries</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;foobar, 2&gt;</td>
<td>a</td>
</tr>
<tr>
<td>&lt;foo, 8&gt;</td>
<td>b</td>
</tr>
<tr>
<td>&lt;bar, 9&gt;</td>
<td>c</td>
</tr>
<tr>
<td>&lt;foobar, 5&gt;</td>
<td>d</td>
</tr>
<tr>
<td>&lt;foo, 8&gt;</td>
<td>b</td>
</tr>
<tr>
<td>&lt;bar, 9&gt;</td>
<td>c</td>
</tr>
</tbody>
</table>

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

crc32 with inlining
BLAST: 348,670,006
OAT: 348,840,008

Whole-program Control-flow Path Attestation
Experimental Setup

Comparison with CFLAT & OAT

BLAST overhead is not even visible!
Performance of BLAST

Whole-program Control-flow Path Attestation
Runtime Overhead Breakdown

![Bar chart showing runtime overhead breakdown for different programs]

- SFI instrumentation
- Ball Larus instrumentation
- Log Hashing

Programs include: aha-mont64, crc32, cubic, edn, huffbench, matmult-int, minver, nbbody, nettle-aes, nettle-sha256, primecourt, sglb-combined, st, tarfind, ud

Whole-program Control-flow Path Attestation
Impact of Reserving Registers

The diagram illustrates the overhead (%) for various benchmarks under different conditions.

- One Register Reserved
- Two Registers Reserved

The benchmarks include: aha-mont64, crc32, cubic, edin, huffbench, matmult-int, minver, nbbody, nettle-aes, nettle-sha256, primecount, sglib-combined, st, tarfind, ud.
# Effectiveness of WPP Representation

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

### Open Syringe Pump Code

<table>
<thead>
<tr>
<th>Code</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;steps; i++) dispenseMedicine();</td>
<td>1 dispenseMedicine(); 8 9 dispenseMedicine();</td>
</tr>
</tbody>
</table>

### WPPs

<table>
<thead>
<tr>
<th>Bolus = 0.010 ml</th>
<th>Execution path trace: 1 8 <em>(repeated 67 times)</em> 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>S -&gt; 1 AAEF 9</td>
<td></td>
</tr>
<tr>
<td>A -&gt; BB</td>
<td></td>
</tr>
<tr>
<td>B -&gt; CC</td>
<td></td>
</tr>
<tr>
<td>C -&gt; DD</td>
<td></td>
</tr>
<tr>
<td>D -&gt; EE</td>
<td></td>
</tr>
<tr>
<td>E -&gt; FF</td>
<td></td>
</tr>
<tr>
<td>F -&gt; 8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bolus = 0.011 ml</th>
<th>Execution path trace: 1 8 <em>(repeated 74 times)</em> 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>S -&gt; 1 AACE 9</td>
<td></td>
</tr>
<tr>
<td>A -&gt; BB</td>
<td></td>
</tr>
<tr>
<td>B -&gt; CC</td>
<td></td>
</tr>
<tr>
<td>C -&gt; DD</td>
<td></td>
</tr>
<tr>
<td>D -&gt; EE</td>
<td></td>
</tr>
<tr>
<td>E -&gt; FF</td>
<td></td>
</tr>
<tr>
<td>F -&gt; 8</td>
<td></td>
</tr>
</tbody>
</table>
## Syringe Pump Benchmark

<table>
<thead>
<tr>
<th>Bolus (mL)</th>
<th>Baseline Time (s)</th>
<th>BLAST Time (s)</th>
<th>BLAST Raw Overhead (s)</th>
<th>CFLAT Raw Overhead (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5 mL</td>
<td>1.28</td>
<td>1.42</td>
<td>0.14 (10%)</td>
<td>1.2 (93%)</td>
</tr>
<tr>
<td>1 mL</td>
<td>2.56</td>
<td>2.71</td>
<td>0.15 (5%)</td>
<td>2.4 (93%)</td>
</tr>
<tr>
<td>2 mL</td>
<td>5.12</td>
<td>5.28</td>
<td>0.16 (3%)</td>
<td>4.8 (93%)</td>
</tr>
</tbody>
</table>
Whole-Program Control-flow Path Attestation

Nikita Yadav and Vinod Ganapathy

ACM Conference on Computer and Communications Security (CCS) 2023

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