Run-time Attestation and Auditing: The Verifier's Perspective

WiSec 2025

Adam Ilyas Caulfield
University of Waterloo

Norrathep Rattanavipanon
Prince of Songkla University
Phuket Campus

Ivan De Oliveira Nunes
University of Zurich

Embedded devices

1. Low-cost, energy efficient MCUs

Embedded devices

1. Low-cost, energy efficient MCUs

2. Remotely deployed to execute safety-critical tasks in modern systems

Embedded devices

1. Low-cost, energy efficient MCUs

2. Remotely deployed to execute safety-critical tasks in modern systems
   a. Sensor/alarm system
   b. Transportation efficiency

Industrial IoT Platform Market Expands with Demand for Predictive Maintenance, and Real-Time Data Analytics

NEWS PROVIDED BY
SNS Insider
November 18, 2024, 13:00 GMT

SHARE THIS ARTICLE

Embedded devices

1. Low-cost, energy efficient MCUs

2. Remotely deployed to execute safety-critical tasks in modern systems
   a. Sensor/alarm system
   b. Transportation efficiency

Industrial IoT Platform Market Expands with Demand for Predictive Maintenance, and Real-Time Data Analytics

NEWS PROVIDED BY
SNS Insider
November 18, 2024, 13:00 GMT

SHARE THIS ARTICLE

SMART Grants Program

Strengthening Mobility and Revolutionizing Transportation (SMART)

The Bipartisan Infrastructure Law (BIL) established the Strengthening Mobility and Revolutionizing Transportation (SMART) discretionary grant program with $100 million appropriated annually for fiscal years (FY) 2022-2026.

SMART

Embedded devices

1. Low-cost, energy efficient MCUs

2. Remotely deployed to execute safety-critical tasks in modern systems
   a. Sensor/alarm system
   b. Transportation efficiency

3. Resource constrained
   a. CPU & Memory
   b. **Security!!!**

4. Still used despite lack of inherent security

Industrial IoT Platform Market Expands with Demand for Predictive Maintenance, and Real-Time Data Analytics

NEWS PROVIDED BY
SNS Insider
November 18, 2024, 13:00 GMT

SHARE THIS ARTICLE

SMART Grants Program

Strengthening Mobility and Revolutionizing Transportation (SMART)

The Bipartisan Infrastructure Law (BIL) established the Strengthening Mobility and Revolutionizing Transportation (SMART) discretionary grant program with $100 million appropriated annually for fiscal years (FY) 2022-2026.

SMART

How can an operator of a remotely deployed MCU verify it is executing in a trustworthy manner?

Remote Static/Binary Attestation (RA):

- Detect software integrity compromise on remote device

Verifier (Vrf)

Prover (Prv)

(1) Send Challenge *chal*

(2) A root of trust (RoT) performs authenticated measurement

$H = Auth_k(chal, PMEM)$

(3) Send *H*

(4) Verify the result

$Verify_k(H, chal, PMEM_{Vrf})$

Remote Static/Binary Attestation (RA):

- Detect software integrity compromise on remote device

Verifier (Vrf)

Prover (Prv)

(1) Send *chal*

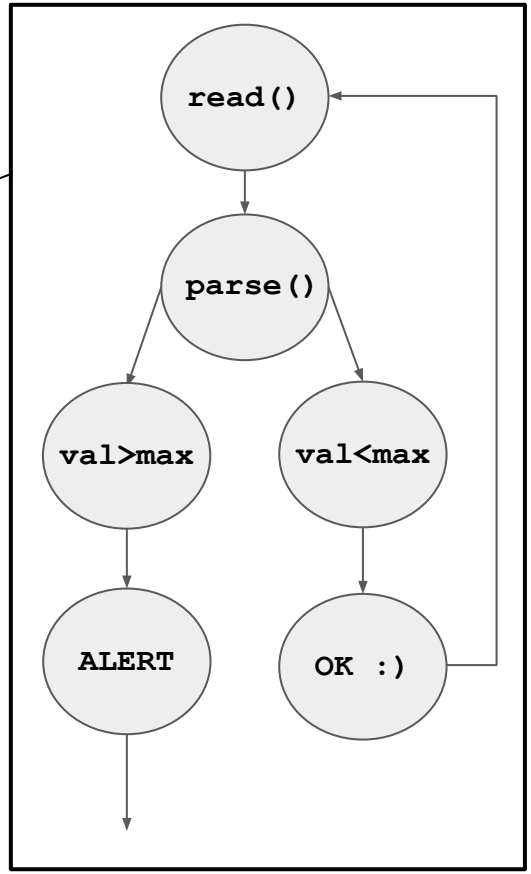
(2) A root of trust (RoT) performs authenticated measurement

$H = Auth_k(chal, PMEM)$

(3) Send *H*

(4) Verify the result

$Verify_k(H, chal, PMEM_{Vrf})$



Remote Static/Binary Attestation (RA):

- Control Flow Hijacks don't modify code, thus go undetected

Verifier (Vrf)

Prover (Prv)

(1) Send *chal*

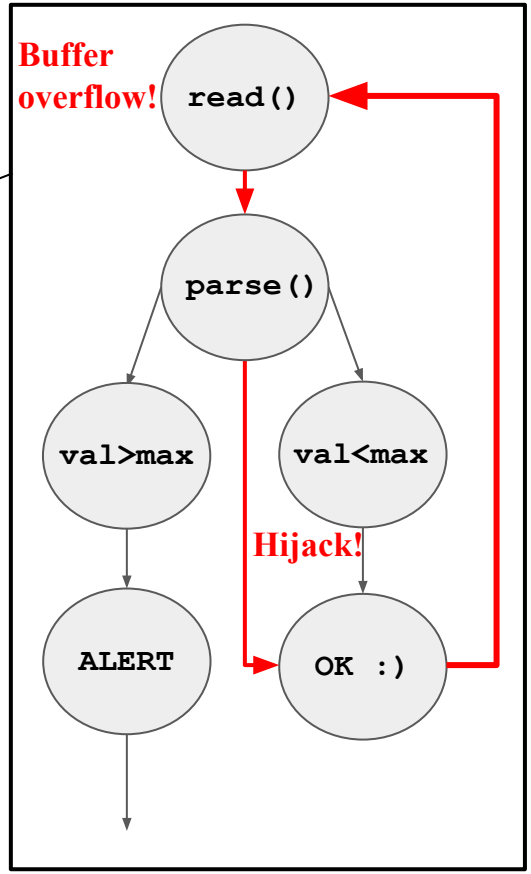
(2) A root of trust (RoT) performs authenticated measurement

$H = Auth_k(chal, PMEM)$

(3) Send ***H***

(4) Verify the result

$Verify_k(H, chal, PMEM_{Vrf})$



Remote Run-time Attestation

- Represent run-time behavior in attestation evidence
- e.g., Control Flow Attestation (CFA)

Verifier (Vrf)

Prover (Prv)

(1) Send *chal*

(2) Execute while RoT traces software

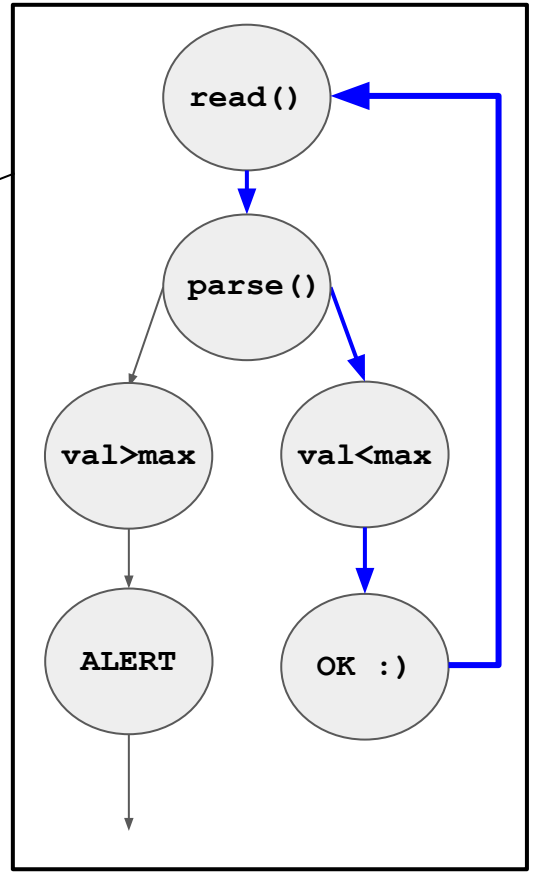
T = `read() → parse() → val<max → OK → …`

(3) RoT performs authenticated measurement

$H = Auth_k(chal, T, PMEM)$

(4) Send H, T

(5) $Verify_k(H, T, chal, PMEM_{Vrf})$



Research in Run-time attestation

First proposal was C-FLAT (CCS ’16)[1]

Many advancements in Prover RoT since then:

- Attesting to data flow and control flow[2,3,4]
- Optimizing run-time evidence[3,4,5,6,]
- From run-time attestation to *auditing*:
  - Guaranteeing eventual evidence delivery and secure remediation action[7,8]

[1] T. Abera et al., “C-FLAT: control-flow attestation for embedded systems software,” in CCS, 2016
[2] I. D. O. Nunes et al., “DIALED: Data Integrity Attestation for Low end Embedded Devices,” in DAC. IEEE, 2021
[3] Z. Sun et al., “OAT: Attesting operation integrity of embedded devices,” in S&P. IEEE, 2020.
[4] G. Dessouky et al., “LiteHAX: lightweight hardware-assisted attestation of program execution,” in ICCAD. IEEE, 2018.
[5] I. D. O. Nunes et al., “Tiny-CFA: A minimalistic approach for control flow attestation using verified proofs of execution,” DATE, 2021.
[6] A. Caulfield et al., “SpecCFA: Enhancing control flow attestation/auditing via application-aware sub-path speculation,” ACSAC 2024
[7] A. Caulfield et al., ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation,” in USENIX Security. USENIX, 2023
[8] A. Caulfield et al., TRACES: Tee-based runtime auditing for commodity embedded systems,” 2024

Research in Run-time attestation

One largely unexplored area: the Verifier's role

Research Questions in this work:

- What do Verifiers actually learn from run-time attestation evidence?
- Can we enable advanced analysis of run-time evidence by Verifier's beyond their role described in typical work?
  - To pinpoint root-cause vulnerabilities, generate patches, and validate them?
- Can we enable end-to-end advanced automated analysis in harsh constraints?
  - E.g,. when no (or partial) source code is available due to use of proprietary software

Run-time Attestation and Auditing from Verifier’s Perspective

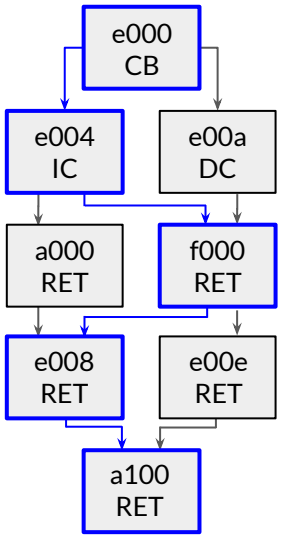
1. Classification of evidence from existing CFA schemes
    a. Their strengths, weaknesses, and trade-offs
    b. A particular focus on their ability to enable automated root-cause analysis

Run-time Attestation and Auditing from Verifier's Perspective

1. Classification of evidence from existing CFA schemes
    a. Their strengths, weaknesses, and trade-offs
    b. A particular focus on their ability to enable automated root-cause analysis

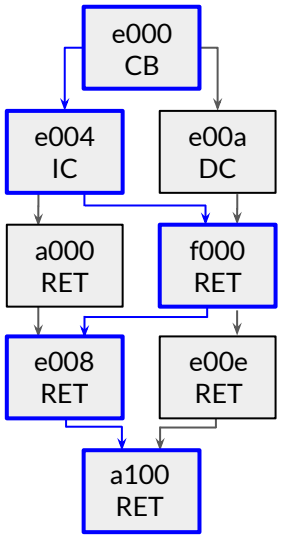
2. Propose **SABRE**
    a. A proof of concept framework based on our findings
    b. Enables a verifier to determine root cause buffer overflow or use-after-free memory vulnerabilities that lead to control flow hijacking
    c. Automatically generates and validates a binary patch
    d. End-to-end process that ***only requires binary executables and attestation evidence***

How do CFA RoT's construct run-time evidence?



Existing works represent run-time behavior in various formats

How do CFA RoT's construct run-time evidence?



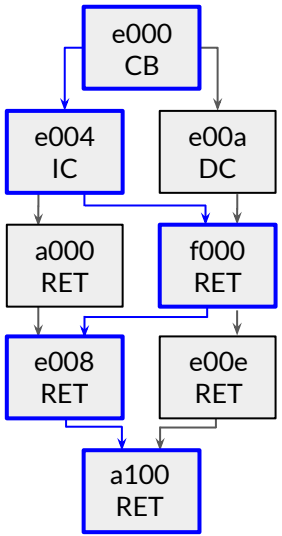
(a) Hash-based

(b) Verbatim Trace

(c) Hybrid

Existing works represent run-time behavior in various formats

How do CFA RoT's construct run-time evidence?



(a) Hash-based

$H_0 = H(e004 \oplus 0)$

$H_1 = H(f000 \oplus H_0)$

$H_2 = H(e008 \oplus H_1)$

$H_{CF} = H(a100 \oplus H_2)$

$E1 = H_{CF}$

(b) Verbatim Trace

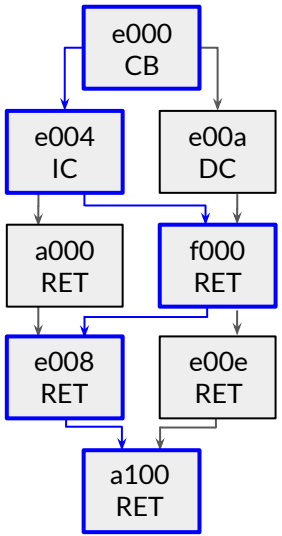
(c) Hybrid

Hash-based[1,9,10,11,12,13]

- Earliest strategy
- Fixed size
- Verification limitations

[9] G. Dessouky et al., "LO-FAT: Low-overhead control flow attestation in hardware," in DAC, 2017
[10] S. Zeitouni et al., "ATRIUM: Runtime attestation resilient under memory attacks," in ICCAD. IEEE, 2017
[11] M. Morbitzer et al., "GuaranTEE: Introducing control-flow attestation for trusted execution environments," CLOUD. IEEE, 2023.
[12] D. Huo et al., "Lape: A lightweight attestation of program execution scheme for bare-metal systems," in HPCC/SmartCity/DSS. IEEE, 2020
[13] M. Geden et al., "Hardware-assisted remote runtime attestation for critical embedded systems," in PST. IEEE, 2019

How do CFA RoT's construct run-time evidence?



(a) Hash-based	(b) Verbatim Trace
$H_0 = H(e004 \oplus 0)$	T = [e004]
$H_1 = H(f000 \oplus H_0)$	T = [e004, f000]
$H_2 = H(e008 \oplus H_1)$	T = [e004, f000, e008]
$H_{CF} = H(a100 \oplus H_2)$	T = [e004, f000, e008, a100]
E1 = H_{CF}	**E2 = T**

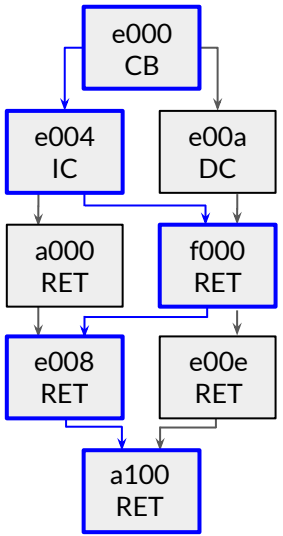
(c) Hybrid

Verbatim Trace[2,5,7,8,14,15,16,17,18]

- Control Flow Log (CF_{Log})
- Alternative to make entire path visible
- Very large evidence
- Limited to small programs[5] or requires optimizations[6]

[14] F. Toffalini et al., "ScaRR: Scalable runtime remote attestation for complex systems," in RAID, 2019.
[15] Y. Zhang et al., "ReCFA: resilient control-flow attestation," in ACSAC, 2021.
[16] N. Yadav et al., "Whole-program control-flow path attestation," in CCS, 2023
[17] A. J. Neto et al., "ISC-FLAT: On the conflict between control flow attestation and real-time operations," in RTAS. IEEE, 2023
[18] M. Ammar et al., "On bridging the gap between control flow integrity and attestation schemes," in USENIX Security, 2024

How do CFA RoT's construct run-time evidence?



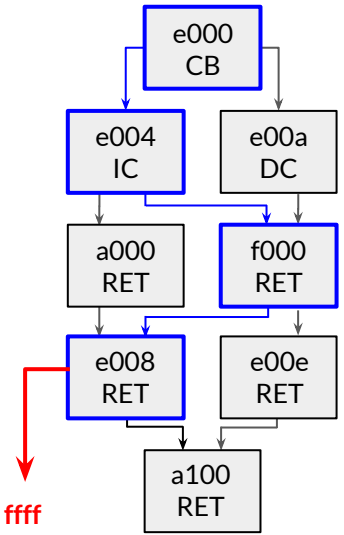
(a) Hash-based	(b) Verbatim Trace	(c) Hybrid
$H_0 = H(e004 \oplus 0)$	T = [e004]	$T_F = [0]$ $H_0 = 0$
$H_1 = H(f000 \oplus H_0)$	T = [e004, f000]	$T_F = [0,f000]$ $H_0 = 0$
$H_2 = H(e008 \oplus H_1)$	T = [e004, f000, e008]	$T_F = [0,f000]$ $H_1 = H(e008 \oplus H_0)$
$H_{CF} = H(a100 \oplus H_2)$	T = [e004, f000, e008, a100]	$T_F = [0,f000]$ $H_R = H(a100 \oplus H_1)$
$E1 = H_{CF}$	**E2 = T**	**$E3 = [T_F, H_R]$**

Hybrid[3,19]

- Mix between the two
- Forward edge is verbatim
- Backward edge is hashed
- Idea is to provide minimal information required to traverse the path

[19] J. Wang et al., "Ari: Attestation of real-time mission execution integrity," USENIX Security, 2023

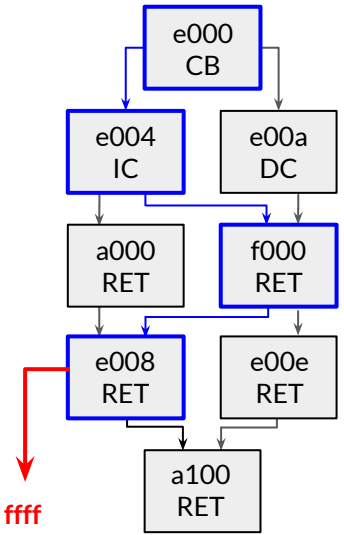
How do CFA RoT's construct run-time evidence?



(a) Hash-based	(b) Verbatim Trace	(c) Hybrid
$H_0 = H(e004 \oplus 0)$	T = [e004]	$T_F = [0]$ $H_0 = 0$
$H_1 = H(f000 \oplus H_0)$	T = [e004, f000]	$T_F = [0,f000]$ $H_0 = 0$
$H_2 = H(e008 \oplus H_1)$	T = [e004, f000, e008]	$T_F = [0,f000]$ $H_1 = H(e008 \oplus H_0)$
$H_{CF} = H(ffff \oplus H_2)$	T = [e004, f000, e008, ffff]	$T_F = [0,f000]$ $H_R = H(ffff \oplus H_1)$
E1 = H_{CF}	**E2 = T**	**E3 = [T_F, H_R]**

What about when something goes wrong?

How do CFA RoT's construct run-time evidence?

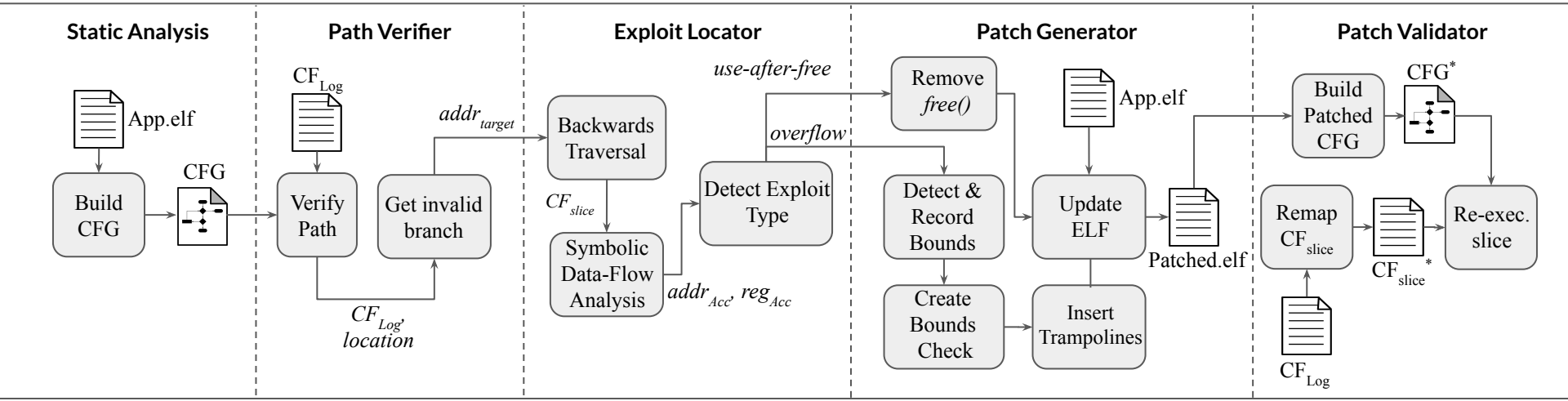


(a) Hash-based	(b) Verbatim Trace	(c) Hybrid
$H_0 = H(e004 \oplus 0)$	T = [e004]	$T_F = [0]$ $H_0 = 0$
$H_1 = H(f000 \oplus H_0)$	T = [e004, f000]	$T_F = [0,f000]$ $H_0 = 0$
$H_2 = H(e008 \oplus H_1)$	T = [e004, f000, e008]	$T_F = [0,f000]$ $H_1 = H(e008 \oplus H_0)$
$H_{CF} = H(ffff \oplus H_2)$	T = [e004, f000, e008, ffff]	$T_F = [0,f000]$ $H_R = H(ffff \oplus H_1)$
E1 = H_{CF}	**E2 = T**	**E3 = [T_F, H_R]**

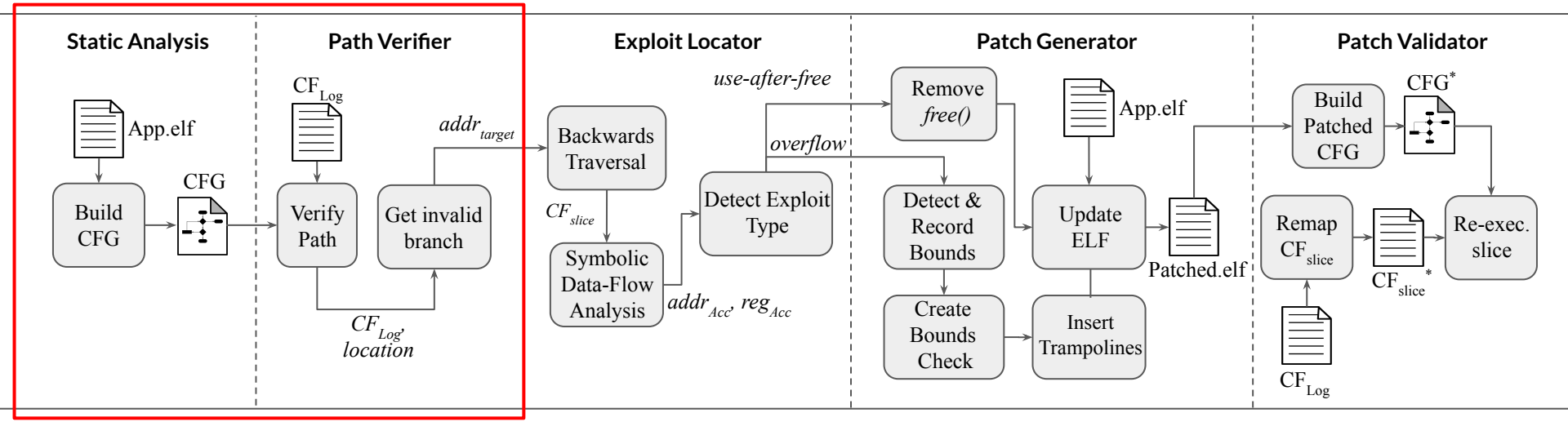
What about when something goes wrong?

Takeaway:
Verbatim trace always reveals the exact corrupted control flow instruction and target address

SABRE: High Level Workflow



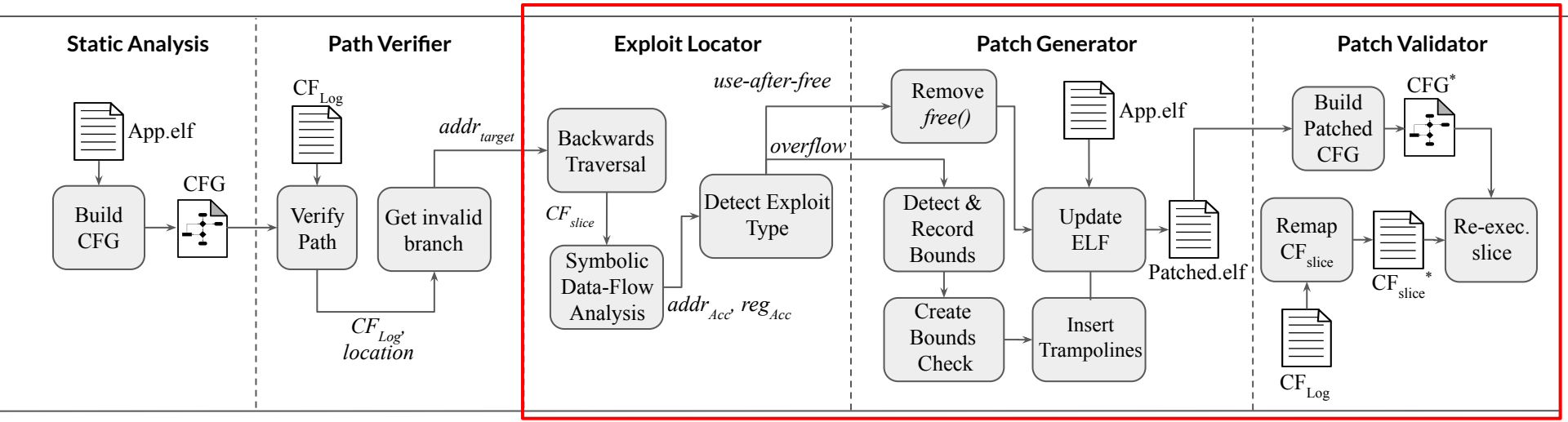
SABRE: High Level Workflow



First two submodules: Static Analysis and Path Verifier

Equivalent steps to prior work

SABRE: High Level Workflow



Novel components: Exploit Locator, Patch Generator, Patch Validator

Inputs: CF_{Log}, App executable

Does not require source code

SABRE: High Level Workflow

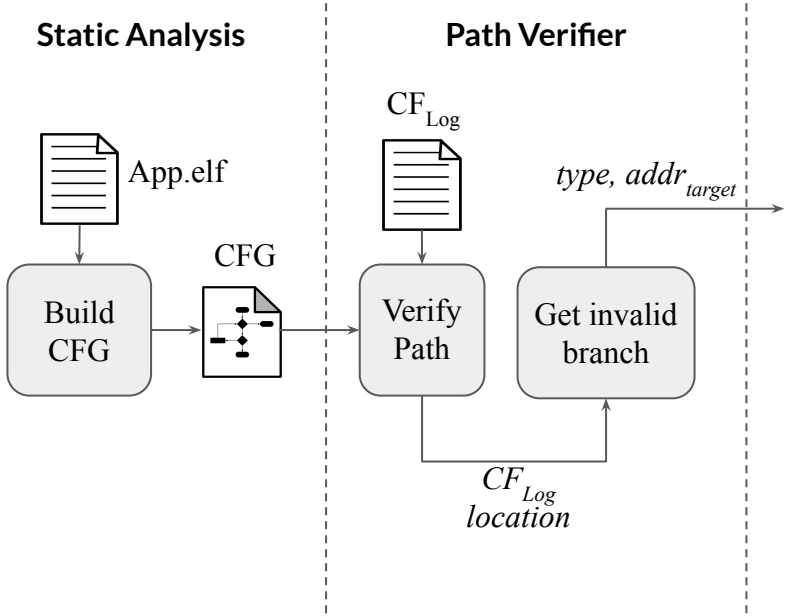
Typical Verifier Capabilities from prior works:

Static Analyzer:

- Builds CFG from App's binary

Path Verifier:

- Executes once CF_{Log} is obtained
- Traverses CFG using the path in CF_{Log}
- Checks for violations using
  - CFG
  - Emulated shadow stack
- Outputs:
  - Exploit type (return address, indr. jump/call)
  - Target address

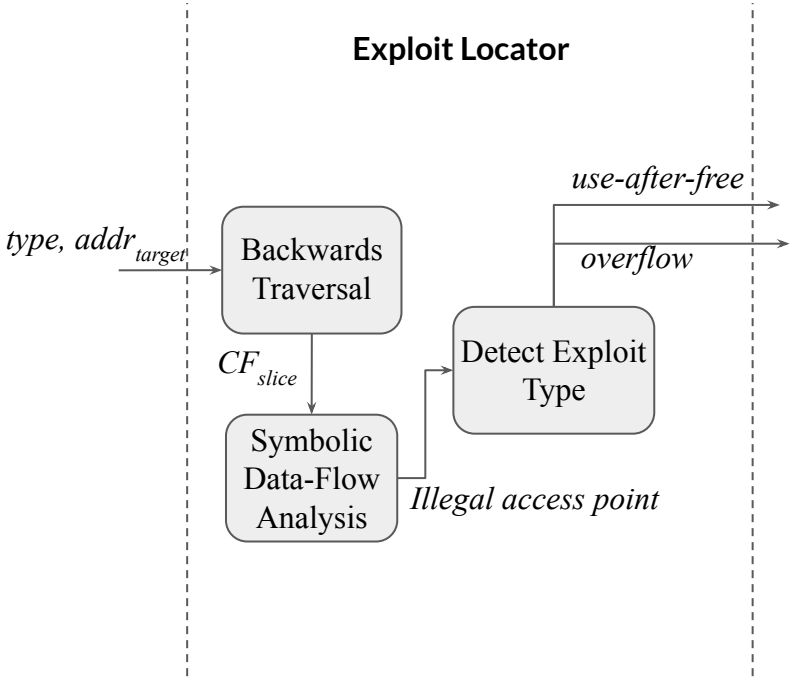


SABRE: High Level Workflow

Exploit Locator

- Goal: pinpoint the instruction that corrupted the target address, and guess the exploit type

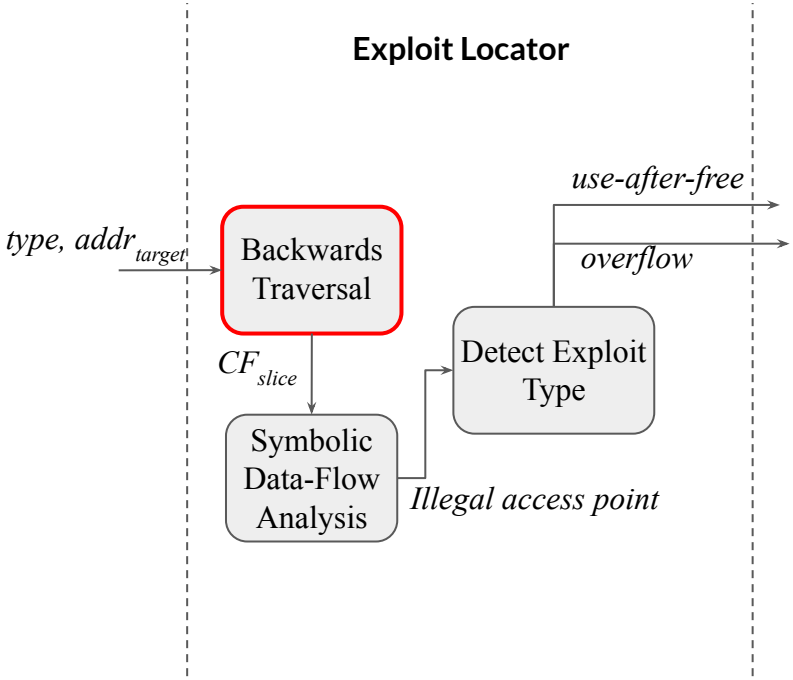
- Two phases:
  - Backwards Traversal
  - Symbolic Data-flow Analysis



SABRE: High Level Workflow

Backwards Traversal

- Uses the CF_{Log} to trace backwards in the App's binary
- Goal: determine the last point during the previous execution where the target address had a valid value

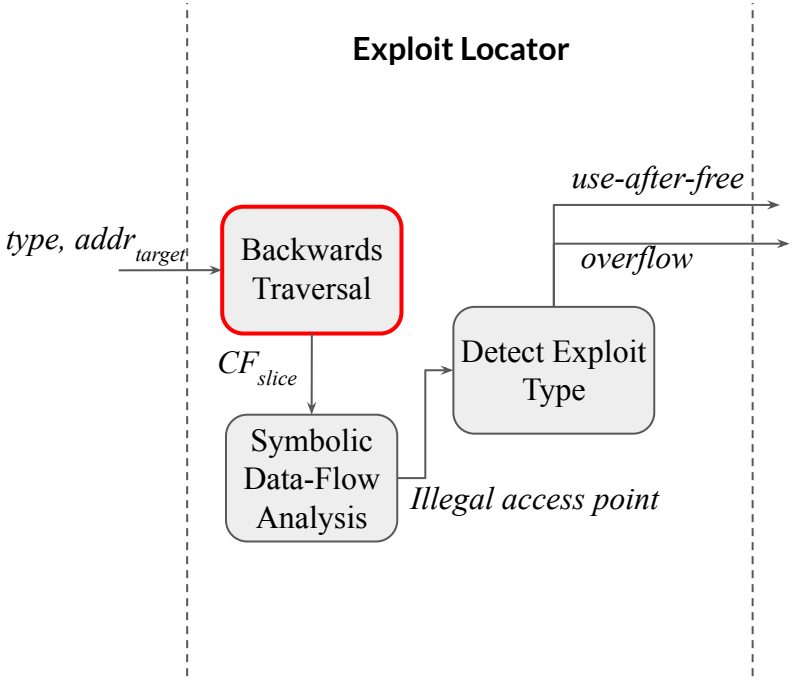


SABRE: High Level Workflow

Backwards Traversal

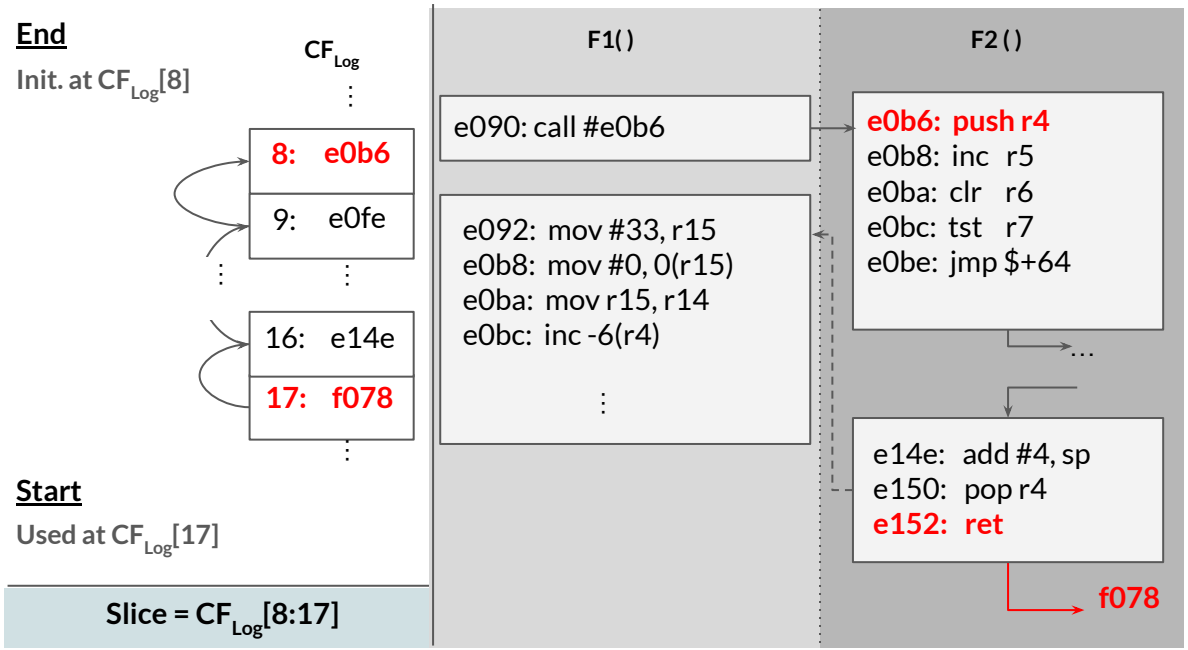
Based on the type of exploit

- Return address
  - looks for matching function call
- Function pointer
  - Tracks definitions to a *“known pointer”*
  - E.g.: malloc output, stack pointer, fixed memory address
- Outputs a slice of CF_{Log} (CF_{slice}) representing the part of the trace that corrupted the target address



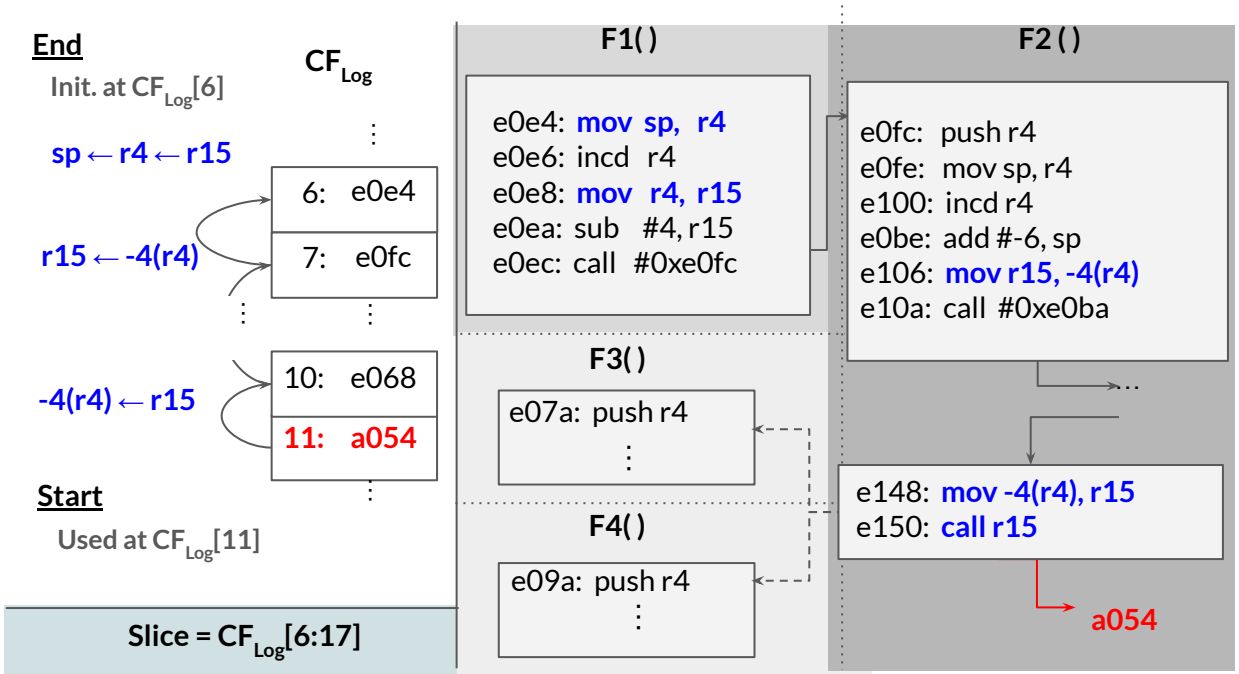
SABRE: High Level Workflow

Backwards Traversal: Return Address example



SABRE: High Level Workflow

Backwards Traversal: Indirect call example

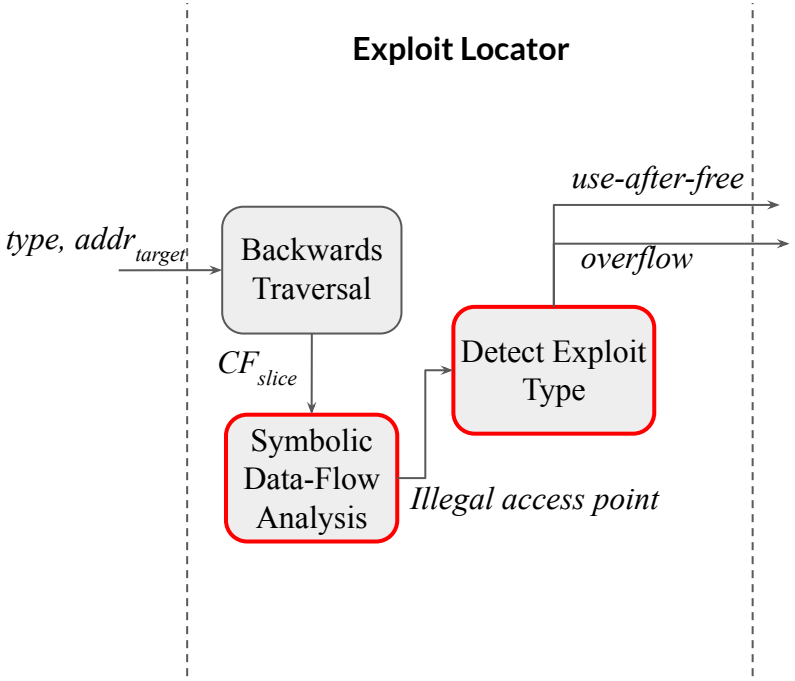


SABRE: High Level Workflow

Symbolic Data-flow Analysis

Procedure:

- Sets the memory address of the target as a special symbol (*X*)
- Evaluates all register & memory operations symbolically
- Stops when the target of memory operation evaluates to *X*
- Outputs the instruction address attempted to modify *X* as the illegal access point



SABRE: High Level Workflow

Symbolic Data-flow Analysis

Procedure:

- Sets the memory address of the target as a special symbol (*X*)
- Evaluates all register & memory operations symbolically
- Stops when the target of memory operation evaluates to *X*
- Outputs the instruction address attempted to modify *X* as the illegal access point

Regs {… , sp ← X, … }

Mem = {...}

For instr in CF_{Slice}:

evaluate(instr)

***Example**: mov r4, addr1*

Get src_{expr} ← Regs[r4]

Get $dest_{expr}$ ← Mem[addr1]

if (simplify($dest_{expr}$) == X) → ABORT!

else mem[$dest_{expr}$] ← simplify(src_{expr})

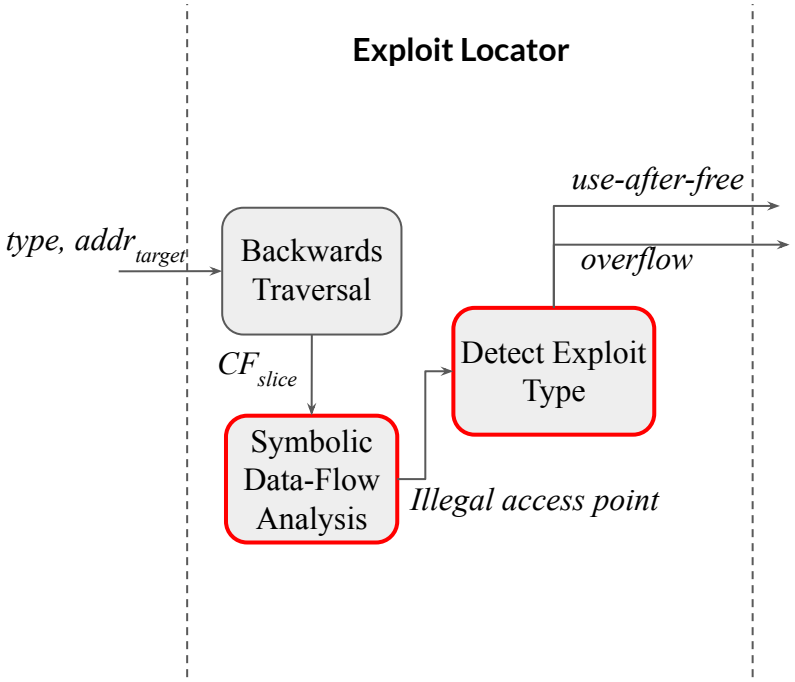
SABRE: High Level Workflow

Symbolic Data-flow Analysis

Detects the exploit type from the symbolic data-flow analysis

Looks for two of the most apparent memory vulnerabilities in embedded systems software:

- Use-after-free, buffer overflows
- based on the state information obtained during the symbolic data-flow analysis.



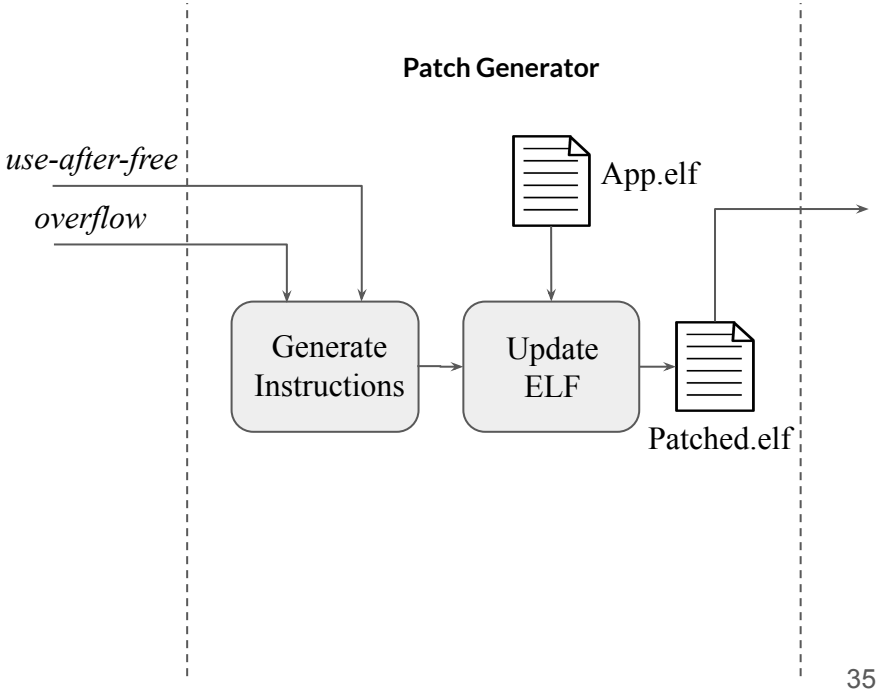
SABRE: High Level Workflow

Patch Generator

Generates patch instructions and updates the binary based on the detected memory vulnerability

Patch instructions depend on the detected vulnerability

- Use-after-free: remove the corrupted *free()*
- Buffer overflow: replace the call to unsafe function with call to a safe function
  - A modified version that checks the buffer bounds

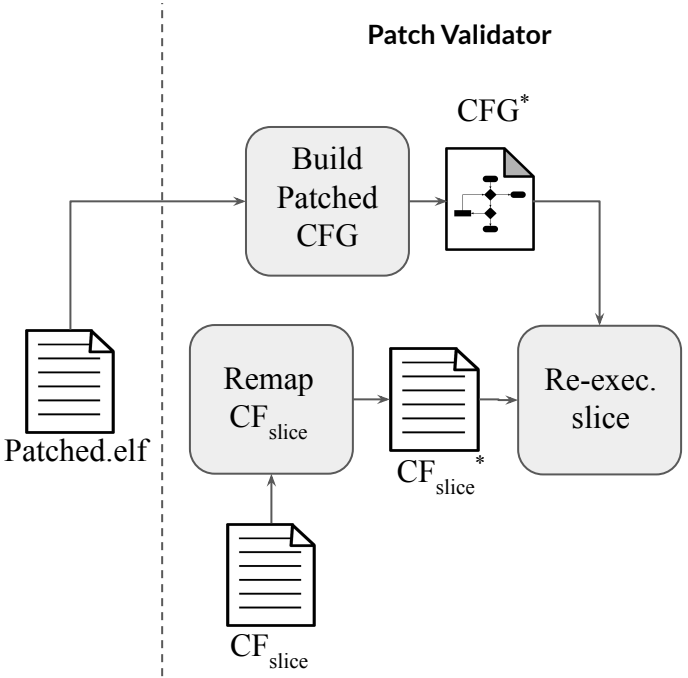


SABRE: High Level Workflow

Patch Validator

Re-evaluates the previous attack trace against the patched binary

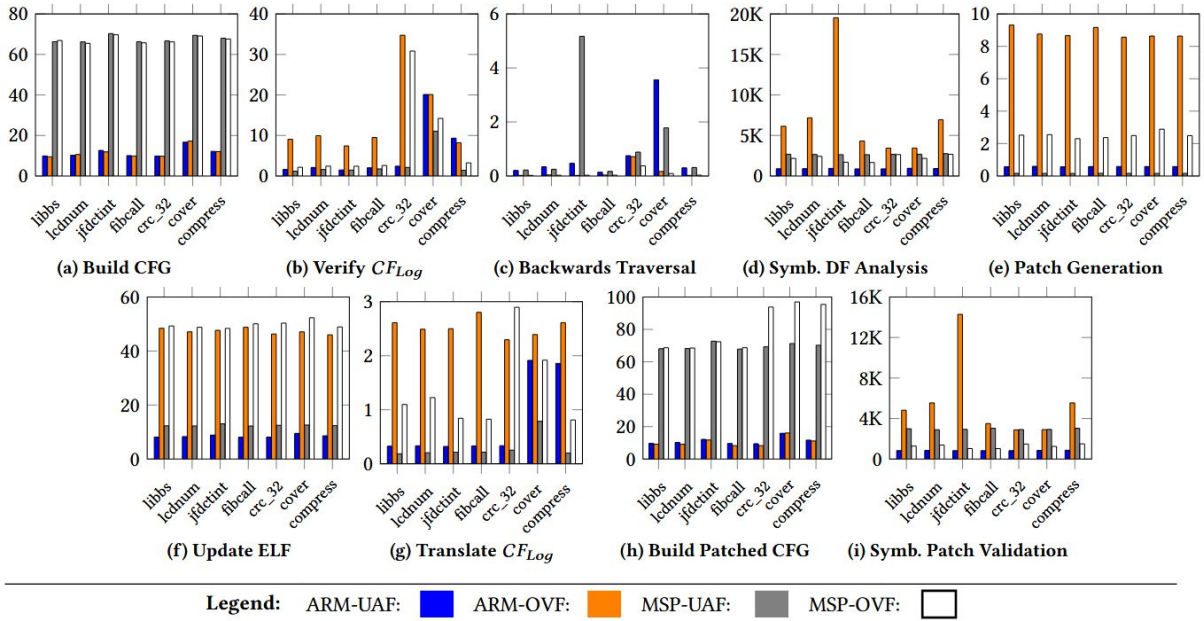
- First builds a new CFG from the patched binary
- Remaps the CF_{slice} to be an equivalent trace from the new binary
- Re-execute the slice (symbolic data-flow analysis)
  - Success: can emulate the previous slice with the new binary
  - Otherwise, generate a report for manual inspection



SABRE: Evaluation

Evaluation

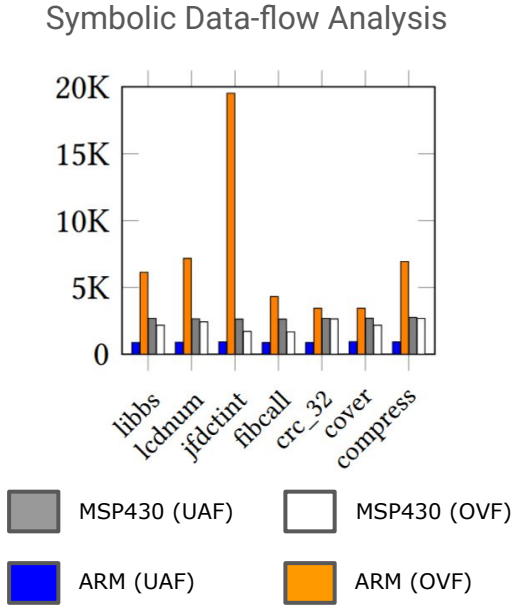
- Craft control flow attacks that exploit use-after-free and buffer overflow vulnerabilities
- Emulate behavior from CVE's by inserting them into benchmark applications
- Evaluate on
  - MSP430+ACFA
  - ARM Cortex-M33+TRACES



SABRE: Evaluation

Timing takeaways

- Symbolic data-flow analysis takes a long time
  - Up to ~19.5 seconds
  - All other modules: < 100 ms



SABRE: Evaluation

Timing takeaways

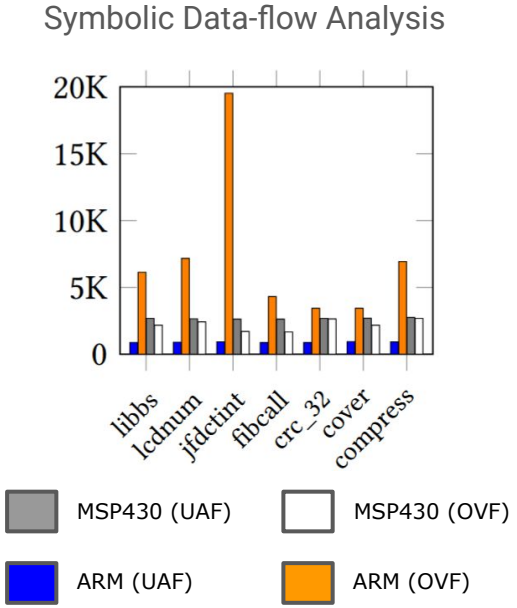
- Symbolic data-flow analysis takes a long time
  - Up to ~19.5 seconds
  - All other modules: < 100 ms

Patch sizes:

- Measured PMEM usage increase before and after the implementing the patch

- Average increase for Use-after-free patch;
  - No overhead
  - Replace free with nops

- Average increase for Buffer Overflow patch:
  - MSP430: ≈264.5 bytes (≈3.31%)
  - ARM: ≈189.7 bytes (≈0.10%)



Summary

- We analyzed the perspective of device operators deploying run-time attestation & auditing protocols over their remotely controlled devices
- Found verbatim-trace-based evidence is most suitable for advanced analysis
- Proposed **SABRE** as a proof of concept framework

Contact: acaulfield@uwaterloo.ca

Resources:

Full paper (preprint):

Open-source prototype: