西安電子科技大學
XIDIAN UNIVERSITY

ReCFA: Resilient Control-Flow Attestation

Yumei Zhang
Xinzhi Liu
Xidian University

Cong Sun
Xidian University
suncong@xidian.edu.cn

Dongrui Zeng
Pennsylvania State University

Gang Tan
Pennsylvania State University

Xiao Kan
Xidian University

Siqi Ma
The University of Queensland

ACSAC 2021

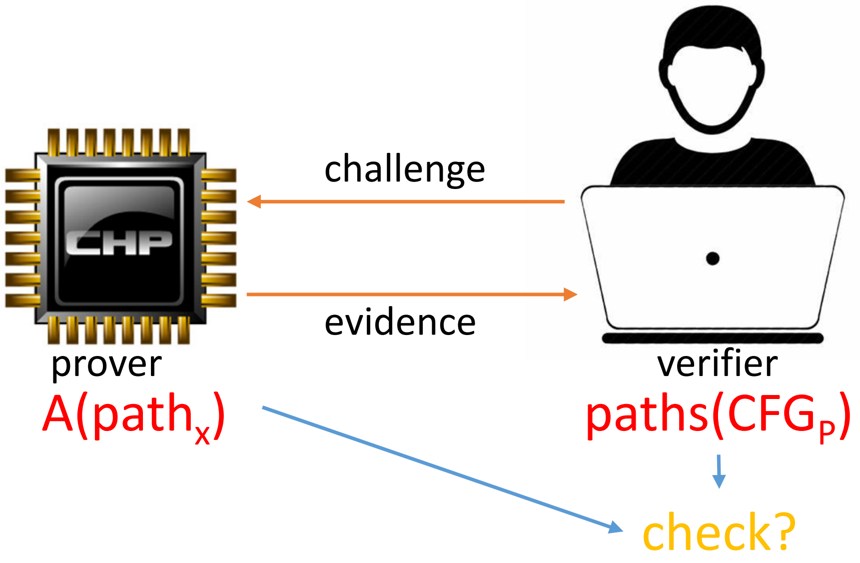
Background and Motivations

- Remote Attestation

Trust Anchor@prover
\+
Attestation protocol

→

Authenticity
\+
Freshness

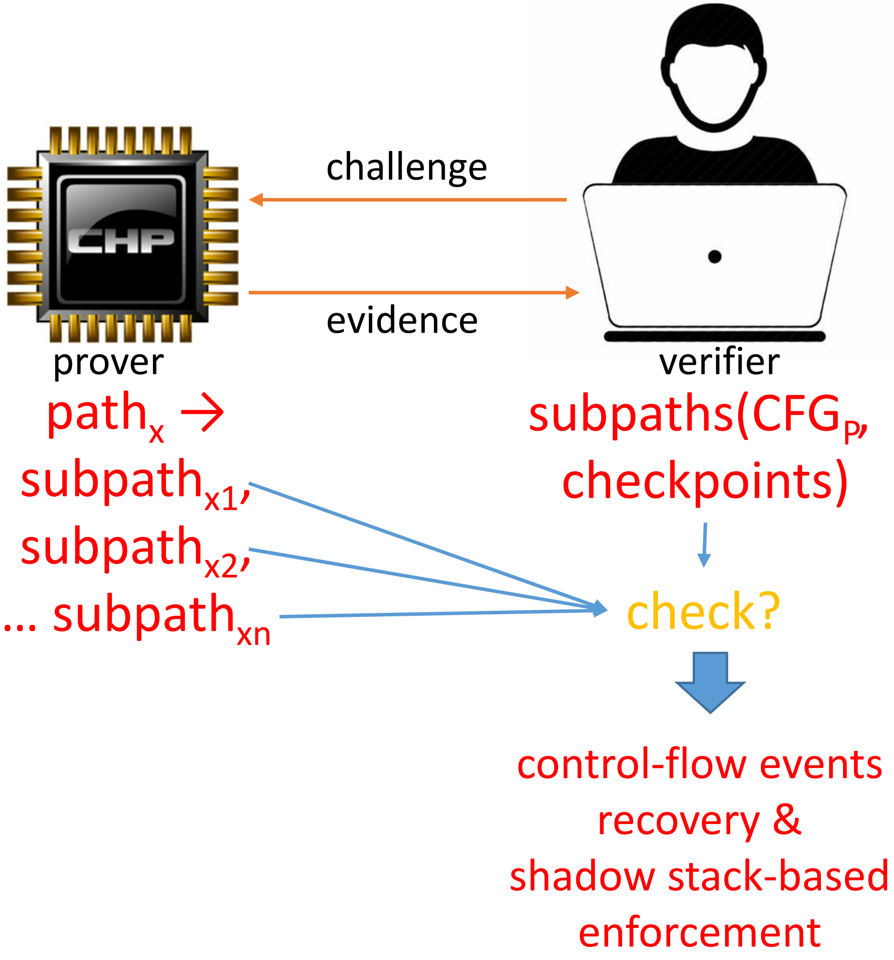


- Control-Flow Attestation (C-FLAT, CCS’16)
  - A kind of runtime attestations.
  - Precisely attest the execution path of the program running at prover.
  - Offline: measure the control-flow paths on CFG and store into measurementDB@verifier
  - Online: measure the executed path@prover as evidence, and check for validity of this path in the measurementDB.

Problem: Complex program → Path explosion when generating measurementDB

Background and Motivations

- Control-flow attestation for complex programs (ScaRR, RAID’19)
  - Mitigate path explosion: measuring checkpoint-separated subpaths.
- Limitations:
  - CFG & measurements generation relies on source code.
  - Measuring checkpoints-separated subpaths causes context missing between subpaths.
  - Coarse-grained path diagnoses. Locate only vulnerable subpath but cannot locate the exact vulnerable control-flow events.



$\text{path}_x \rightarrow \text{subpath}_{x1}, \text{subpath}_{x2}, \ldots \text{subpath}_{xn}$

$\text{subpaths}(\text{CFG}_P, \text{checkpoints})$

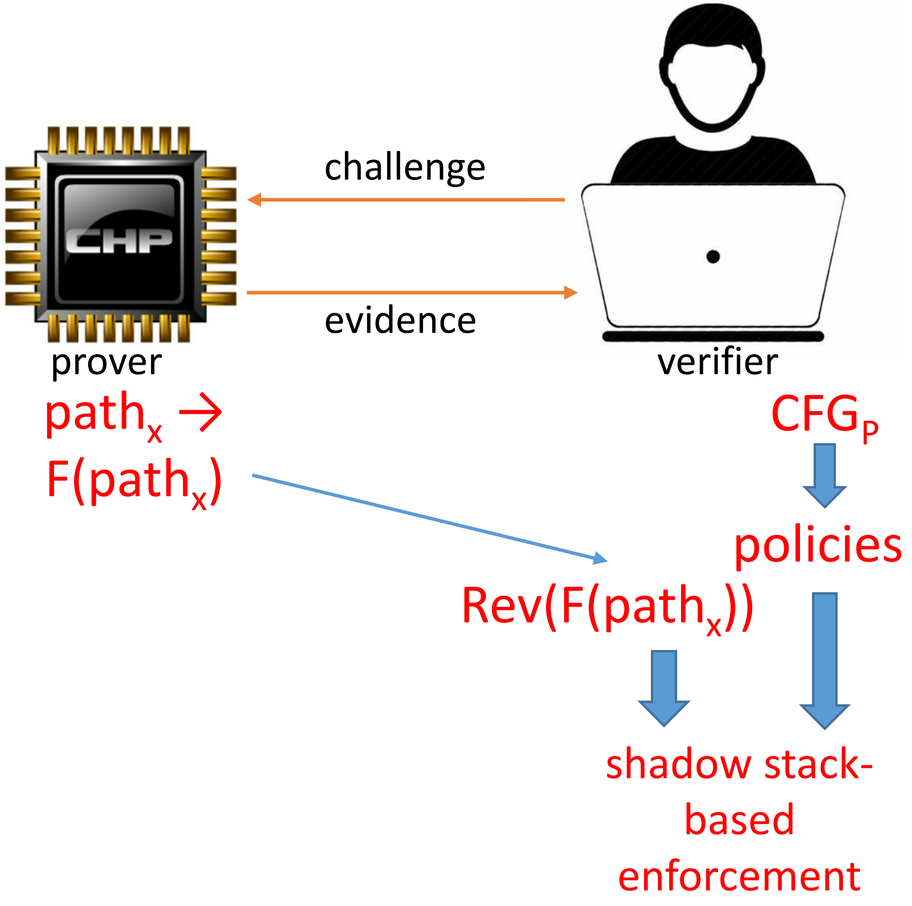
check?

control-flow events recovery & shadow stack-based enforcement

Background and Motivations

- Ideas of ReCFA
  - No offline measurements generation, only binary CFG generated as policy.
  - No source code requirement: binary rewriting of program@prover

- Difficulties:
  - Prover-side events explosion. Require careful design of condensing function **F(•)**.
  - Acceptable runtime overhead at prover. Technical difficulty on rewriting an efficient binary P.



$path_x$ → $F(path_x)$

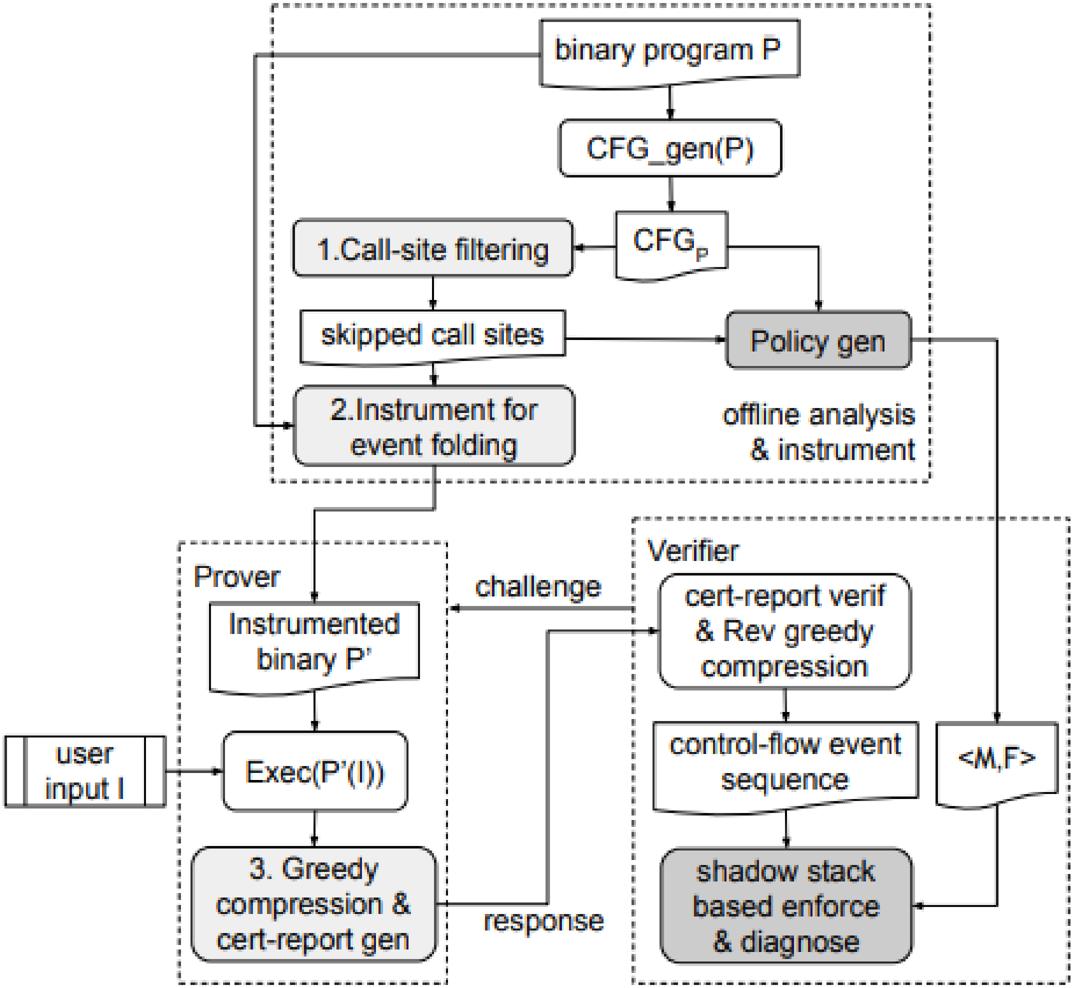
CFG_P

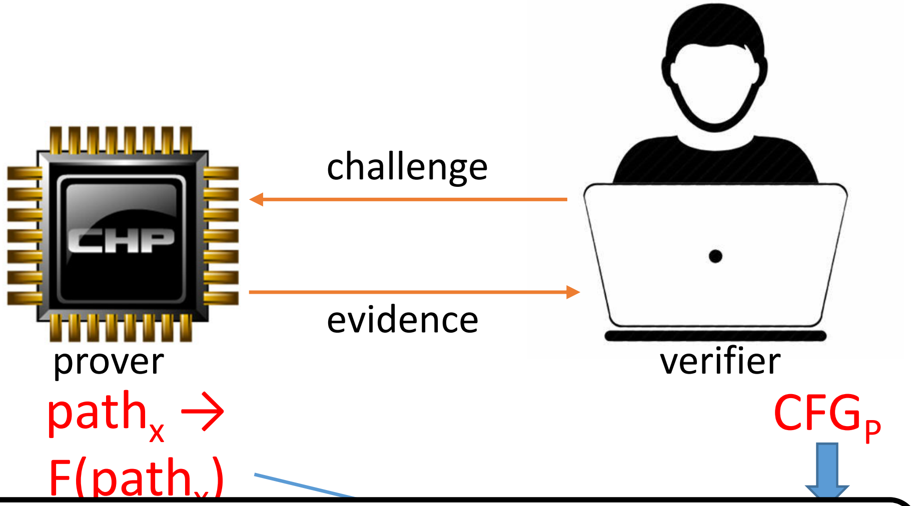
policies

$Rev(F(path_x))$

shadow stack-based enforcement

Design of ReCFA





- Runtime path condensing F(•) is conducted by the instrumented code snippets
- Not every control-flow event has to be instrumented (for efficiency) —— call site filtering
- Further compression on the control-flow event sequence before sending report —— greedy compression.

Threat Model and Requirements (Similar to C-FLAT, ScaRR)

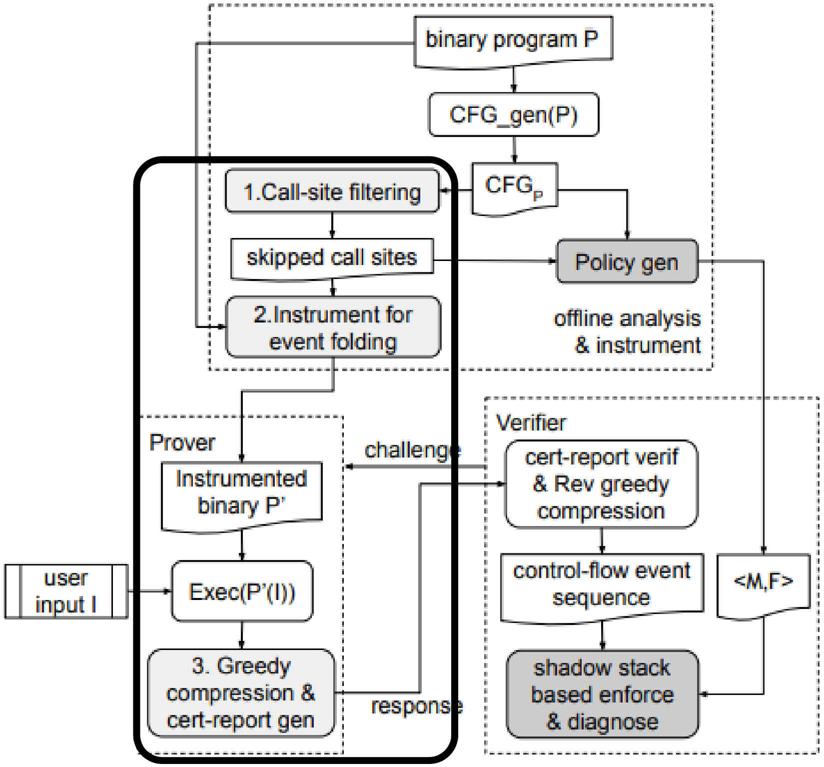
- Assumptions
  - DEP & trust anchor deployed on prover.
  - Off-the-shelf attestation protocol (out of our scope)

- Attackers can
  - run the program with arbitrary input.
  - read/write the data section of the program.
  - exploit memory corruptions to hijack control flow.

- The verifier remotely diagnoses control-flow path leading to control-flow hijacking. (different from local CFI) —— usually higher runtime overhead.

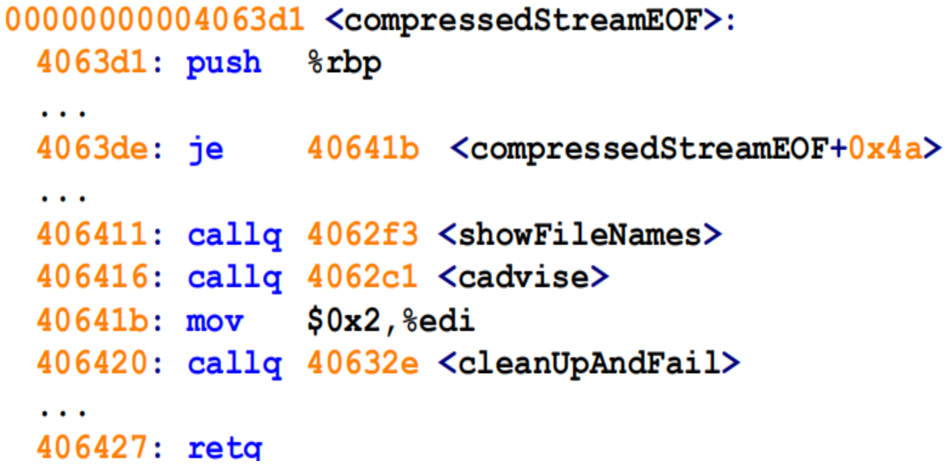
Multi-phase Control-Flow Condensing of ReCFA

- Phase-1: Filter out the skippable direct calls
- Phase-2: Runtime control-flow events folding
- Phase-3: Greedy compression on control-flow event sequence

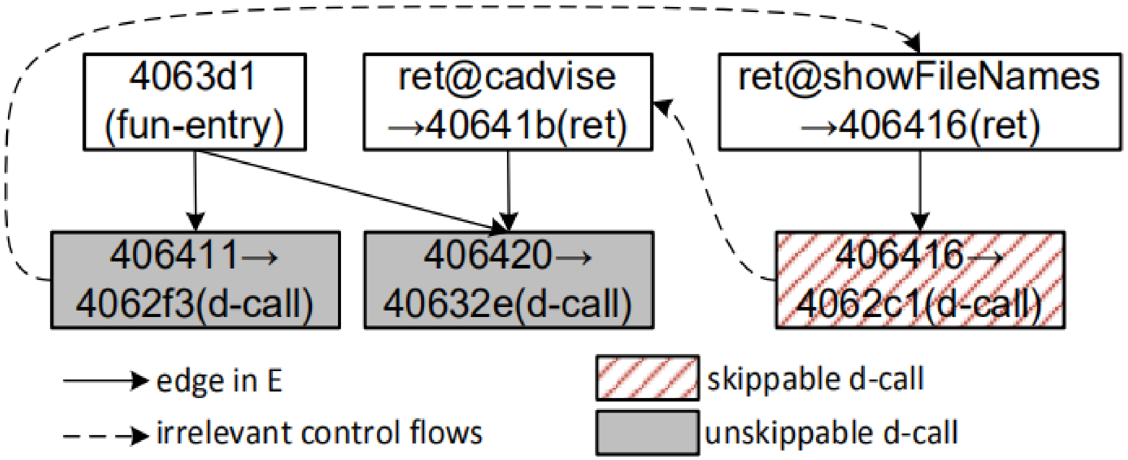


Multi-phase Control-Flow Condensing of ReCFA

- Phase-1: Filter out the skippable direct calls
  - Potential Monitoring Points (PMPs): all function calls, indirect jumps, and returns
  - Intuition: causality relation between consecutive PMPs —— A node is skippable only when none of its predecessors has more than one successor.
  - Build abstract graph from CFG (PMPs as nodes)
  - Detect skippable PMPs (direct calls), only unskippable PMPs are instrumented.
  - Build a mapping M to hold the relation between predecessor and skippable successor (Let the verifier know the skippable node from predecessor node)



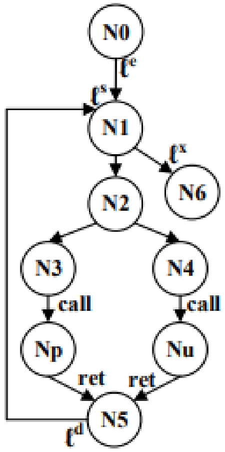
```
00000000004063d1 <compressedStreamEOF>:
  4063d1: push   %rbp
  ...
  4063de: je     40641b  <compressedStreamEOF+0x4a>
  ...
  406411: callq  4062f3 <showFileNames>
  406416: callq  4062c1 <cadvise>
  40641b: mov    $0x2,%edi
  406420: callq  40632e <cleanUpAndFail>
  ...
  406427: retq
```



Multi-Phase Control-Flow Condensing of ReCFA

- Phase-2: Runtime control-flow events folding
  - Instrumented binary code snippets take action
  - We design
    - where and what to be instrumented
    - what data structure to be manipulated for the events folding
  - Folding to capture the unskipped control-flow events in loops and recursions
    - Path explosion mainly caused by loops and recursions

```
N0,N1: for(int i=0; i<n; i++){
N2:      if(i%2==0){
N3:          privileged();
N4:      else unprivileged();
N5:      endif
      }
N6: ...
Np: privileged() {...}
Nu: unprivileged() {...}
```



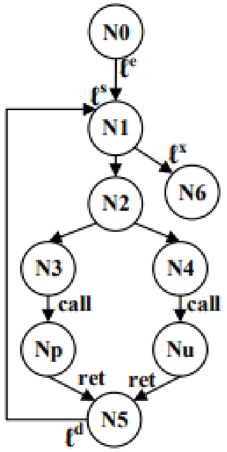
(a) Loop Example

- loop entry (ℓ^e)
- loop exit (ℓ^x)
- loop body start (ℓ^s)
- loop body end (ℓ^d)

Multi-Phase Control-Flow Condensing of ReCFA

- @loop entry: push ⊥ onto loop stack to demarcate outer/inner loop
- @loop body start: start a new stack frame and push its index onto loop stack
- @loop body end
  - compare the top stack frame with the stack frames indexed by the loop stack elements above the top-most ⊥
  - pop the top stack frame and its index when duplicated event path found
- @loop exit: pop the content of loop stack above top-most ⊥, to fold the outer loop

```
N0,N1: for(int i=0; i<n; i++){
N2:      if(i%2==0){
N3:          privileged();
N4:      else unprivileged();
N5:      endif
       }
N6:    ...
Np:  privileged() {...}
Nu:  unprivileged() {...}
```



(a) Loop Example

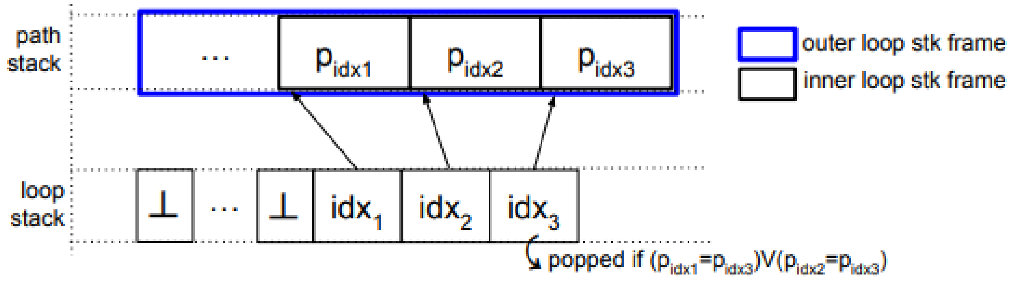


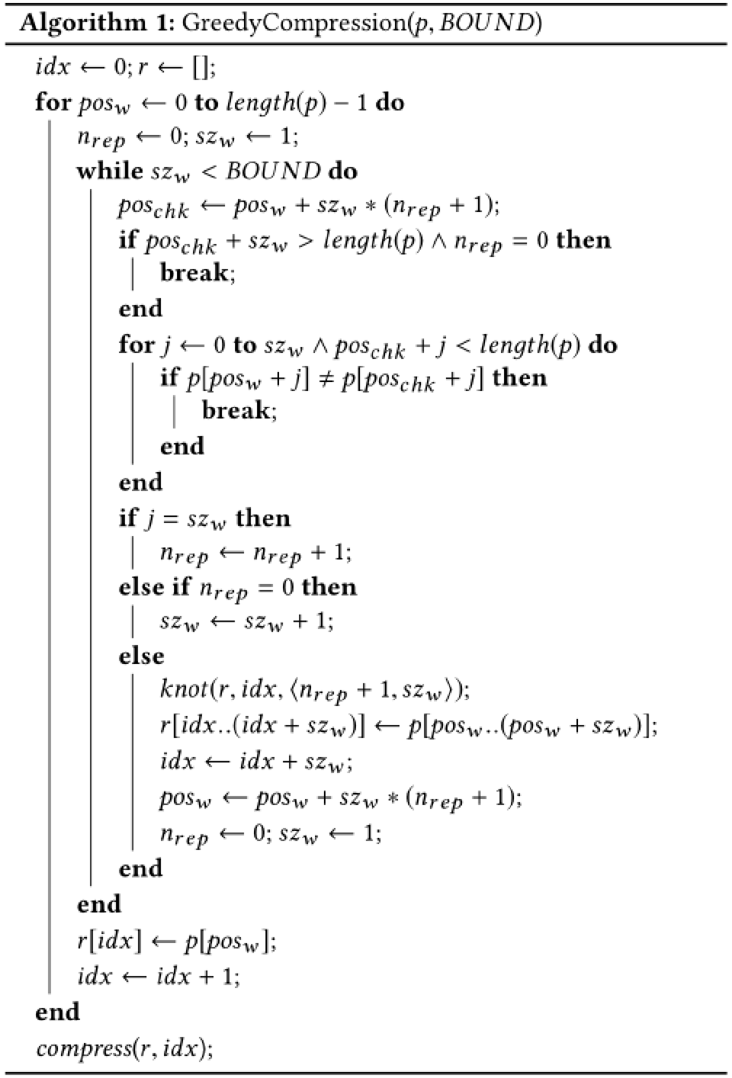
Figure 3: Folding Nested Loops

Multi-Phase Control-Flow Condensing of ReCFA

- Phase-2: Runtime control-flow events folding
  - Use the same data structure as loop stack (i.e. conceptually recursion stack) to deal with recursions
  - Use static analysis to identify the recursion cases causing false positives. Skip folding these cases.

Multi-Phase Control-Flow Condensing of ReCFA

- Phase-3: Greedy compression
  - Irrelevant to program structure. On control-flow events sequence
  - Greedy algorithm with a sliding window
    - Add knot information about repeating times
  - Complexity: O(n*BOUND)
    - n: length of events sequence
    - BOUND: size of sliding window
  - Not optimal:
    - $e1e2e1e2e3e1e2e1e2e3$ compressed to $\langle 2, 2\rangle e1e2e3\langle 2, 2\rangle e1e2e3$ instead of $\langle 2, 5\rangle e1e2e1e2e3$



Algorithm 1: GreedyCompression(p, $BOUND$)

```
$idx \leftarrow 0; r \leftarrow []$;
for $pos_w \leftarrow 0$ to $length(p) - 1$ do
    $n_{rep} \leftarrow 0; sz_w \leftarrow 1$;
    while $sz_w < BOUND$ do
        $pos_{chk} \leftarrow pos_w + sz_w * (n_{rep} + 1)$;
        if $pos_{chk} + sz_w > length(p) \wedge n_{rep} = 0$ then
            break;
        end
        for $j \leftarrow 0$ to $sz_w \wedge pos_{chk} + j < length(p)$ do
            if $p[pos_w + j] \neq p[pos_{chk} + j]$ then
                break;
            end
        end
        if $j = sz_w$ then
            $n_{rep} \leftarrow n_{rep} + 1$;
        else if $n_{rep} = 0$ then
            $sz_w \leftarrow sz_w + 1$;
        else
            $knot(r, idx, \langle n_{rep} + 1, sz_w \rangle)$;
            $r[idx..(idx + sz_w)] \leftarrow p[pos_w..(pos_w + sz_w)]$;
            $idx \leftarrow idx + sz_w$;
            $pos_w \leftarrow pos_w + sz_w * (n_{rep} + 1)$;
            $n_{rep} \leftarrow 0; sz_w \leftarrow 1$;
        end
    end
    $r[idx] \leftarrow p[pos_w]$;
    $idx \leftarrow idx + 1$;
end
$compress(r, idx)$;
```

Context-Sensitive Remote Enforcement

- Verifier-side shadow stack
- Mapping F
    - statically for forward edges. The element of F is in form $cs \mapsto (ca, tgts)$.
    - cs : call site address of a forward edge
    - ca : address of the call-after point of the call site
    - $tgts$: the set of valid target addresses of the call
- Security policy : <M,F>
- For call edge
    - Retrieve the mapping M to find all the skipped events led by this call edge
- For forward edge and its subsequent skipped events
    - Validate the call/branch target (in $tgts$?)
    - Push the call-after point onto the shadow stack
- For returns
    - check "return target =? top element of shadow stack"

Implementation of ReCFA

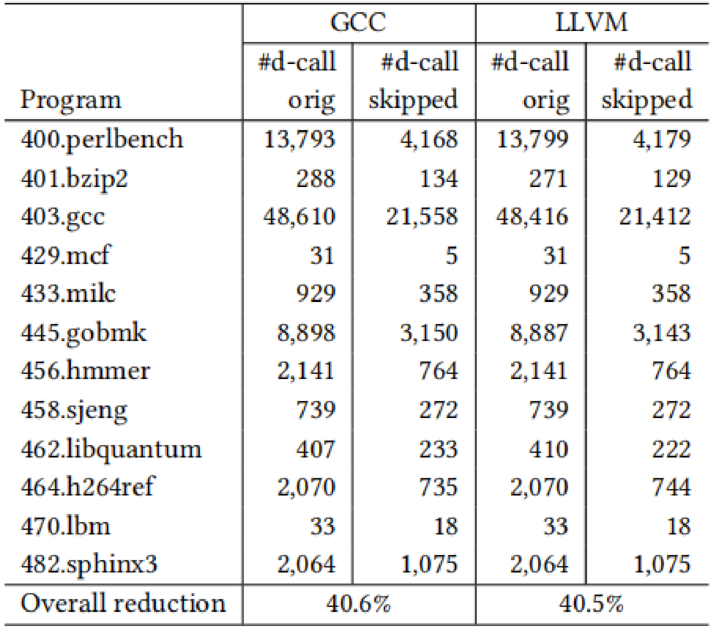
- Binary-level CFG
  - Derived with TypeArmor. Neutral to different binary CFG generation approaches
- Security policy <M,F>
  - M: static analysis with Dyninst
  - F: static analysis with TypeArmor
- Edge encoding
  - Indirect branches and returns: a pair of code addresses
  - Direct call: one code address of the call site
- Intel’s MPK protected user-space data structures (loop stack and path stack)
  - CFA data regions only allowed to be written by instrumented code snippets
  - Insert guards at entry and exit points of code snippet
    - The guard notifies the kernel the type of each snippet and the guarded point
  - Kernel-level pairing the consecutive entry/exit signal of guards with the same snipped type
  - Avoid using indirect branches in the code snippets

Evaluations

- SPEC CPU 2006's C benchmarks (standard workload "test")
- Binaries build with GCC v7.5.0 and LLVM v10.0.0

Effect of call-site filtering

The ratio of reduction ranges 16.1%~57.2% for GCC binaries and 16.1%~54.5% for LLVM binaries. The overall reduction is around 40.5%.



	GCC		LLVM	
Program	#d-call orig	#d-call skipped	#d-call orig	#d-call skipped
400.perlbench	13,793	4,168	13,799	4,179
401.bzip2	288	134	271	129
403.gcc	48,610	21,558	48,416	21,412
429.mcf	31	5	31	5
433.milc	929	358	929	358
445.gobmk	8,898	3,150	8,887	3,143
456.hmmer	2,141	764	2,141	764
458.sjeng	739	272	739	272
462.libquantum	407	233	410	222
464.h264ref	2,070	735	2,070	744
470.lbm	33	18	33	18
482.sphinx3	2,064	1,075	2,064	1,075
Overall reduction	40.6%		40.5%	

Effect of control-flow events folding

Average time overhead of instrumented program is 42.3%

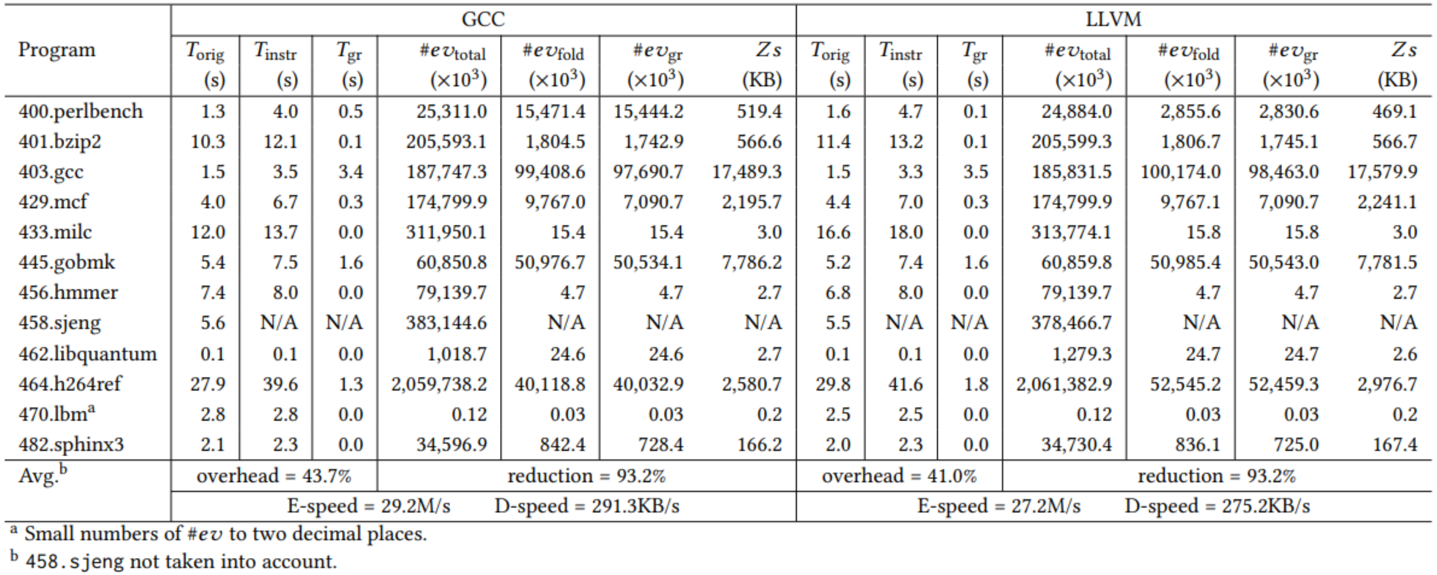
Overall reduction in the control-flow events is 93.2%

Average attestation speed (E-speed) is 28.2M/s

Peak D-speed is 2.53MB/s (GCC) and 2.59MB/s (LLVM). Average D-speed is 283.0KB/s

E-speed: speed of the prover generating raw runtime control-flow events

D-speed: speed of the prover generating data that are sent to the verifier



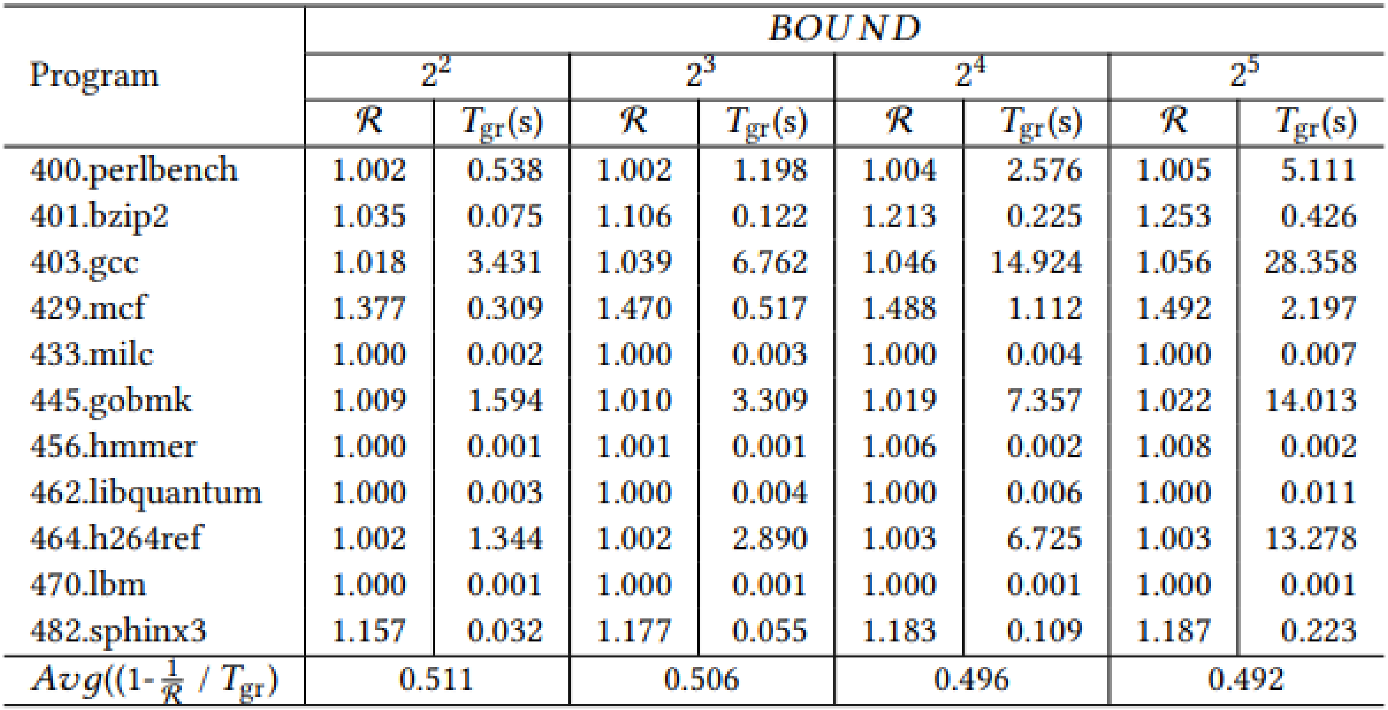
Program	GCC							LLVM						
	T_{orig} (s)	T_{instr} (s)	T_{gr} (s)	$\#ev_{total}$ ($\times 10^3$)	$\#ev_{fold}$ ($\times 10^3$)	$\#ev_{gr}$ ($\times 10^3$)	Zs (KB)	T_{orig} (s)	T_{instr} (s)	T_{gr} (s)	$\#ev_{total}$ ($\times 10^3$)	$\#ev_{fold}$ ($\times 10^3$)	$\#ev_{gr}$ ($\times 10^3$)	Zs (KB)
400.perlbench	1.3	4.0	0.5	25,311.0	15,471.4	15,444.2	519.4	1.6	4.7	0.1	24,884.0	2,855.6	2,830.6	469.1
401.bzip2	10.3	12.1	0.1	205,593.1	1,804.5	1,742.9	566.6	11.4	13.2	0.1	205,599.3	1,806.7	1,745.1	566.7
403.gcc	1.5	3.5	3.4	187,747.3	99,408.6	97,690.7	17,489.3	1.5	3.3	3.5	185,831.5	100,174.0	98,463.0	17,579.9
429.mcf	4.0	6.7	0.3	174,799.9	9,767.0	7,090.7	2,195.7	4.4	7.0	0.3	174,799.9	9,767.1	7,090.7	2,241.1
433.milc	12.0	13.7	0.0	311,950.1	15.4	15.4	3.0	16.6	18.0	0.0	313,774.1	15.8	15.8	3.0
445.gobmk	5.4	7.5	1.6	60,850.8	50,976.7	50,534.1	7,786.2	5.2	7.4	1.6	60,859.8	50,985.4	50,543.0	7,781.5
456.hmmer	7.4	8.0	0.0	79,139.7	4.7	4.7	2.7	6.8	8.0	0.0	79,139.7	4.7	4.7	2.7
458.sjeng	5.6	N/A	N/A	383,144.6	N/A	N/A	N/A	5.5	N/A	N/A	378,466.7	N/A	N/A	N/A
462.libquantum	0.1	0.1	0.0	1,018.7	24.6	24.6	2.7	0.1	0.1	0.0	1,279.3	24.7	24.7	2.6
464.h264ref	27.9	39.6	1.3	2,059,738.2	40,118.8	40,032.9	2,580.7	29.8	41.6	1.8	2,061,382.9	52,545.2	52,459.3	2,976.7
470.lbm[a]	2.8	2.8	0.0	0.12	0.03	0.03	0.2	2.5	2.5	0.0	0.12	0.03	0.03	0.2
482.sphinx3	2.1	2.3	0.0	34,596.9	842.4	728.4	166.2	2.0	2.3	0.0	34,730.4	836.1	725.0	167.4
Avg.[b]	overhead = 43.7%			reduction = 93.2%				overhead = 41.0%			reduction = 93.2%			
	E-speed = 29.2M/s D-speed = 291.3KB/s							E-speed = 27.2M/s D-speed = 275.2KB/s						

[a] Small numbers of #ev to two decimal places.

[b] `458.sjeng` not taken into account.

Effect of *BOUND* value tunning

Greedy compression time increases exponentially along with the exponential increase of $BOUND$. The increase in the gain of compression is not exponential. Thus small $BOUND$ is preferred

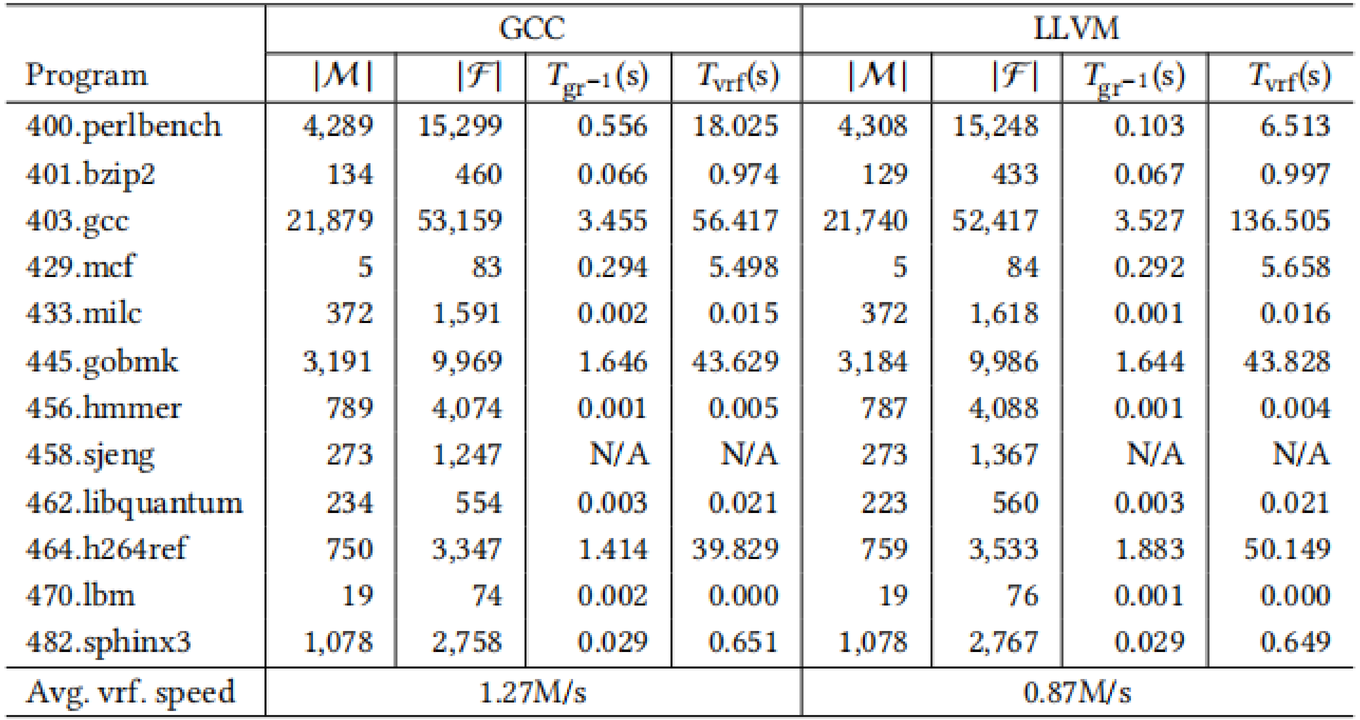


Program	$BOUND$							
	2^2		2^3		2^4		2^5	
	$\mathcal{R}$	$T_{gr}(s)$	$\mathcal{R}$	$T_{gr}(s)$	$\mathcal{R}$	$T_{gr}(s)$	$\mathcal{R}$	$T_{gr}(s)$
400.perlbench	1.002	0.538	1.002	1.198	1.004	2.576	1.005	5.111
401.bzip2	1.035	0.075	1.106	0.122	1.213	0.225	1.253	0.426
403.gcc	1.018	3.431	1.039	6.762	1.046	14.924	1.056	28.358
429.mcf	1.377	0.309	1.470	0.517	1.488	1.112	1.492	2.197
433.milc	1.000	0.002	1.000	0.003	1.000	0.004	1.000	0.007
445.gobmk	1.009	1.594	1.010	3.309	1.019	7.357	1.022	14.013
456.hmmer	1.000	0.001	1.001	0.001	1.006	0.002	1.008	0.002
462.libquantum	1.000	0.003	1.000	0.004	1.000	0.006	1.000	0.011
464.h264ref	1.002	1.344	1.002	2.890	1.003	6.725	1.003	13.278
470.lbm	1.000	0.001	1.000	0.001	1.000	0.001	1.000	0.001
482.sphinx3	1.157	0.032	1.177	0.055	1.183	0.109	1.187	0.223
$Avg((1-\frac{1}{\mathcal{R}}\ /\ T_{gr})$	0.511		0.506		0.496		0.492	

Effectiveness of Context-Sensitive Enforcement at Verifier

The average verification speed is 1.03M/s

Incomparable to the speeds of ScaRR. Different definitions of control-flow events

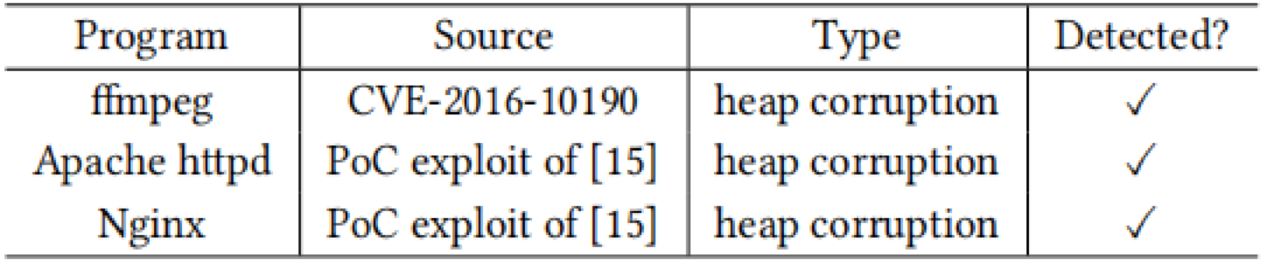


Program	GCC				LLVM			
	$\lvert\mathcal{M}\rvert$	$\lvert\mathcal{F}\rvert$	$T_{\mathrm{gr}^{-1}}$(s)	T_{vrf}(s)	$\lvert\mathcal{M}\rvert$	$\lvert\mathcal{F}\rvert$	$T_{\mathrm{gr}^{-1}}$(s)	T_{vrf}(s)
400.perlbench	4,289	15,299	0.556	18.025	4,308	15,248	0.103	6.513
401.bzip2	134	460	0.066	0.974	129	433	0.067	0.997
403.gcc	21,879	53,159	3.455	56.417	21,740	52,417	3.527	136.505
429.mcf	5	83	0.294	5.498	5	84	0.292	5.658
433.milc	372	1,591	0.002	0.015	372	1,618	0.001	0.016
445.gobmk	3,191	9,969	1.646	43.629	3,184	9,986	1.644	43.828
456.hmmer	789	4,074	0.001	0.005	787	4,088	0.001	0.004
458.sjeng	273	1,247	N/A	N/A	273	1,367	N/A	N/A
462.libquantum	234	554	0.003	0.021	223	560	0.003	0.021
464.h264ref	750	3,347	1.414	39.829	759	3,533	1.883	50.149
470.lbm	19	74	0.002	0.000	19	76	0.001	0.000
482.sphinx3	1,078	2,758	0.029	0.651	1,078	2,767	0.029	0.649
Avg. vrf. speed	1.27M/s				0.87M/s			

Real exploits diagnosed by ReCFA

ReCFA’s verifier detects typical exploits detectable by TypeArmor.

Only instrument on a related part of CFG due to the large size of binary



Program	Source	Type	Detected?
ffmpeg	CVE-2016-10190	heap corruption	✓
Apache httpd	PoC exploit of [15]	heap corruption	✓
Nginx	PoC exploit of [15]	heap corruption	✓

Available: https://github.com/suncongxd/ReCFA

THANKS

Thanks for listening