Adaptive Call-site Sensitive Control Flow Integrity

Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou †, Yueqiang Cheng ‡

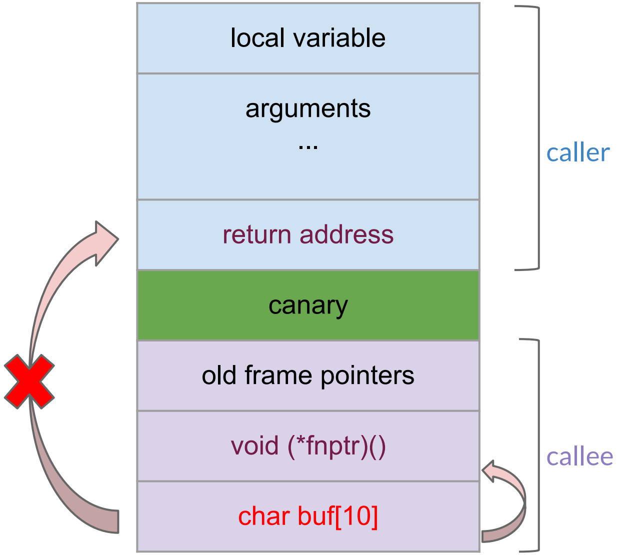
Dept. of Computer Science, Florida State University, Tallahassee, USA
† School of Computer Science, Zhejiang University, Hangzhou, China
‡ Baidu X-lab, Sunnyvale, USA

Control Flow Integrity

Control Flow Integrity (CFI) is a ***defense*** mechanism ***against control-flow hijacking*** that employs inline reference monitor to enforce the run-time control flow of a process must follow the statically computed control-flow graph (CFG).

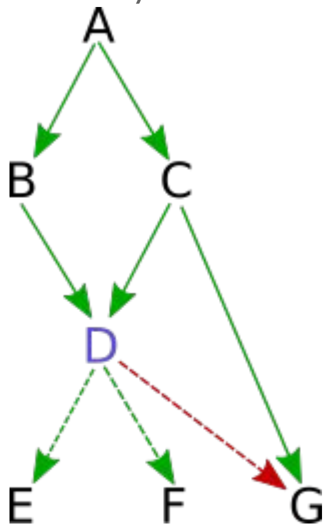
CFI consists of:

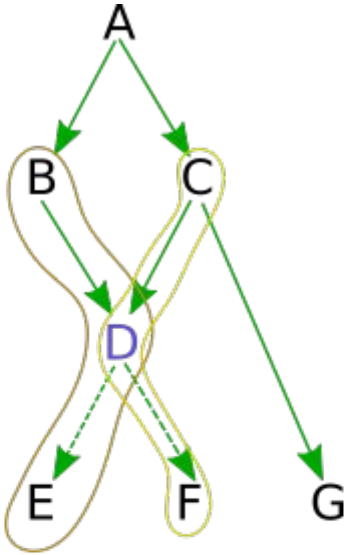
- CFI Policy
- Inline Reference Monitor
- CFG



CFI Policy

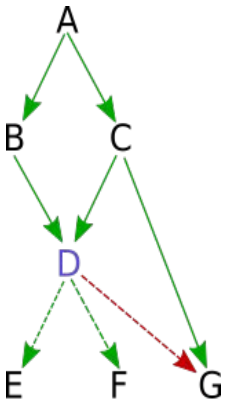
- Context-insensitive (CI-) CFI: CFI policy without additional information
- Context-sensitive (CS-) CFI: CFI policy with past execution history
    - e.g., path sensitivity



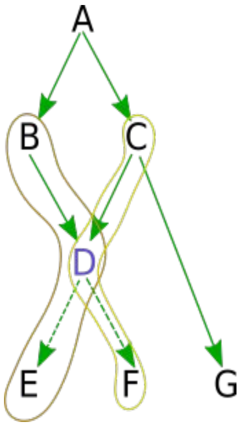


CFI Metrics

- **Equivalence Class (EC)**: A group of targets that CFI cannot distinguish.
- **Largest EC (LC):** Largest allowed targets among all EC’s.



CI-CFI
# of EC = 1 (D)
EC Size = 2 (D->E,F)
LC = 2 (D->E,F)



CS-CFI
# of EC = 2 ({B,D},{C,D})
EC Size = 1 ({B,D}->E, {C,D}->F)
LC = 1 ({B,D}->E, {C,D}->F)

Quantifying CFI

To quantify security guarantee of CFI, we propose:

$$QS_{CFI} = AVG_{EC} \times LC$$

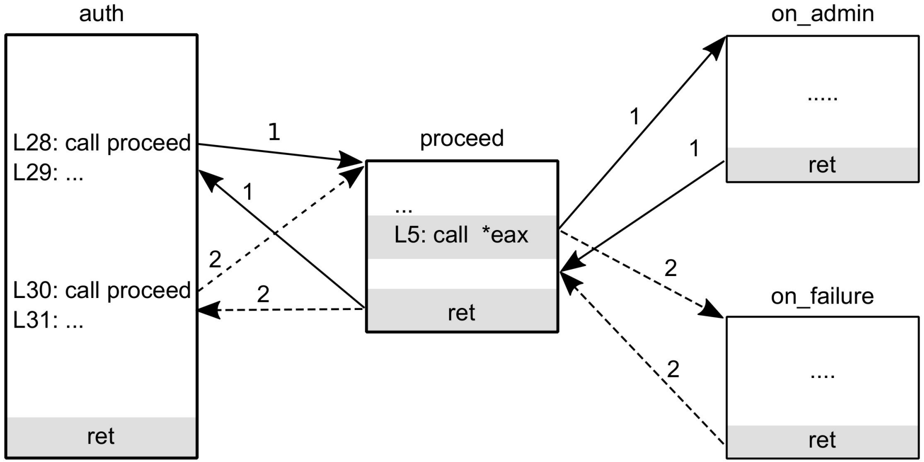
Benefits:

- QS_{CFI} has a theoretical limit of 1, i.e., every target can be individually distinguished and validated.
- Applicable to both context-sensitive and insensitive CFI systems
  - AVG_{EC} avoids the false impression of improvements because context-sensitive CFI can exponentially increase the # of ECs

Call-site Sensitive CFI

At runtime, combine backward edge information (return addresses) with intended target at indirect forward edge call-point:

- By accessing call stack (protected by shadow stack, Intel CET etc.).
- Validates multi-level call-sites against a pre-computed CFG.
- Benefits:
  - Strict CFI policy
  - No extra memory
  - No specialized hardware required
  - Whole-program protection



1. {L29,L5} => on_admin
2. {L31,L5} => on_failure

Challenges

- Context-sensitivity is ***expensive***.
    - Runtime reference monitor:
        - Complex verification method.
        - Extra instrumentation required for integrity.
- Context-sensitivity is ***complicated***.
    - Unavailable scalable static points-to analysis to compute CFG:
        - Precision.
        - Completeness.

CFI-LB: Control Flow Integrity with Look Back

- Strong CFI Policy:
  - Call-site sensitivity.
  - SafeStack for secure call stack.
- Fast and Secure CFI Enforcement:
  - Adaptive call-site depth.
  - Hash-table based set membership test.
  - Intel TSX for guaranteed atomicity.
- Complete and precise CFG:
  - Multi-scope CFG
    - Dynamic-colcolic CFG (CS-CFI)
    - Static CFG (CI-CFG)
  - Localized concolic execution

Adaptive Call-site Sensitivity

Sizes	Call-site(0)	Call-site(1)	Call-site(2)	Call-site(3)
1	149	674	2372	5423
2	27	54	223	667
3	10	18	48	172
4	7	17	35	70
5-10	20	26	56	93
11-20	4	4	28	37
21-40	2	1	0	0
54	1	1	1	1
Total	220	795	2763	6463

TABLE I: EC distribution over sizes of the target sets (403.gcc)

- Individual call-point has independent level of call-site sensitivity.
  - TABLE I: 149 call-points have best security guarantee without any context.
  - TABLE II: LC of 403.gcc is call-site insensitive.

Call-site Depth	Max. Target Set Size	# of Indirect Calls	# of ECs	Avg. EC Size
0	54	161	161	1.48
1	20	23	262	1.37
2	17	36	787	1.5
Total		220	1210	1.47

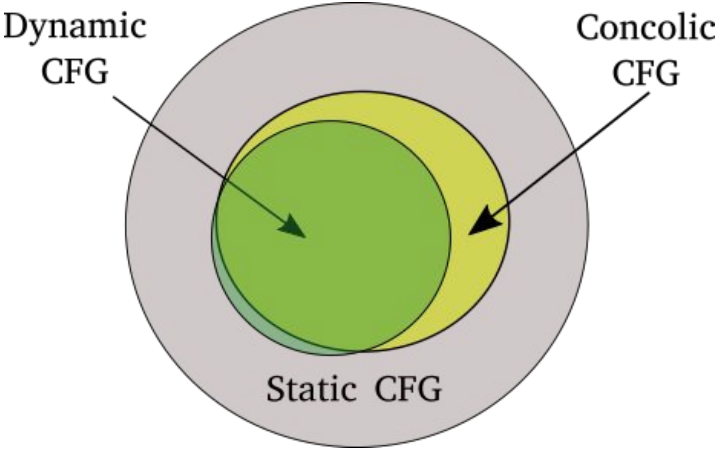
TABLE II: Distribution of adaptive call-site sensitivity.(403.gcc)

Atomicity of Reference Monitor

- Complex monitoring system may cause internal states spill to stack.
- In the multi-threaded programming environment, race condition is critical.
- Solution:
  - Encapsulates reference monitor with Intel TSX hardware transactional memory protection.
    - CFI-LB uses restricted transactional memory (RTM) interface.
  - If a race condition is detected, CFI-LB retries the execution.
    - Transaction could fail without race condition because of cache conflicts

Multi-scope CFG

- Why?
  - Unavailability of scalable context-sensitive static points-to analysis.
  - Completeness is crucial.
  - Imprecision is unacceptable.
- Dynamic CS-CFG
  - Seed input derived.
- Concolic CS-CFG
  - Localized concolic execution.
- Static CI-CFG
  - Act as a fail-safe for dynamic-concolic CFG.
  - In release binary, a control transfer validated by static CFG will be logged.

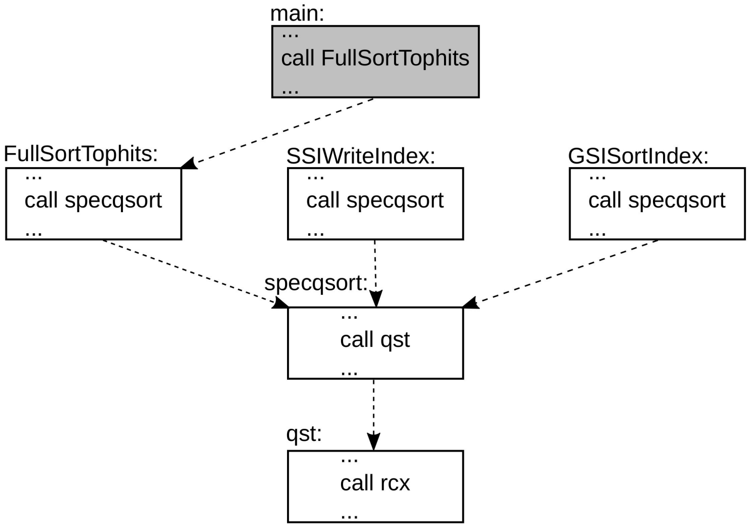


Dynamic CFG

- Due to limited seed input:
    - Dynamic execution can cover limited number of paths.
- Possible solution:
    - Symbolic execution based complete path exploration.
    - Issues with symbolic execution:
        - Constraint solver is slow.
        - Path explosion problem.

Localized Concolic Execution

- For ***n*** call-site sensitivity:
  - Coverage starts from every function in ***n*** call-site depth for each indirect call-point.
  - Localized code coverage: all-possible path reachable to an indirect call-point.
  - Intuition: Call-site sensitive code pointers must have assigned within localized coverage.
- Reconstruct memory layout:
  - Utilize concrete execution memory dump (Concolic).
  - Symbolize memories (except references).
    - Global variables.
    - Arguments at coverage entry points.
    - I/O memories in libc function.
- Benefits:
  - Localized all-possible path coverage *(Scalable)*.
  - Concolic data-driven path exploration *(Precision)*.



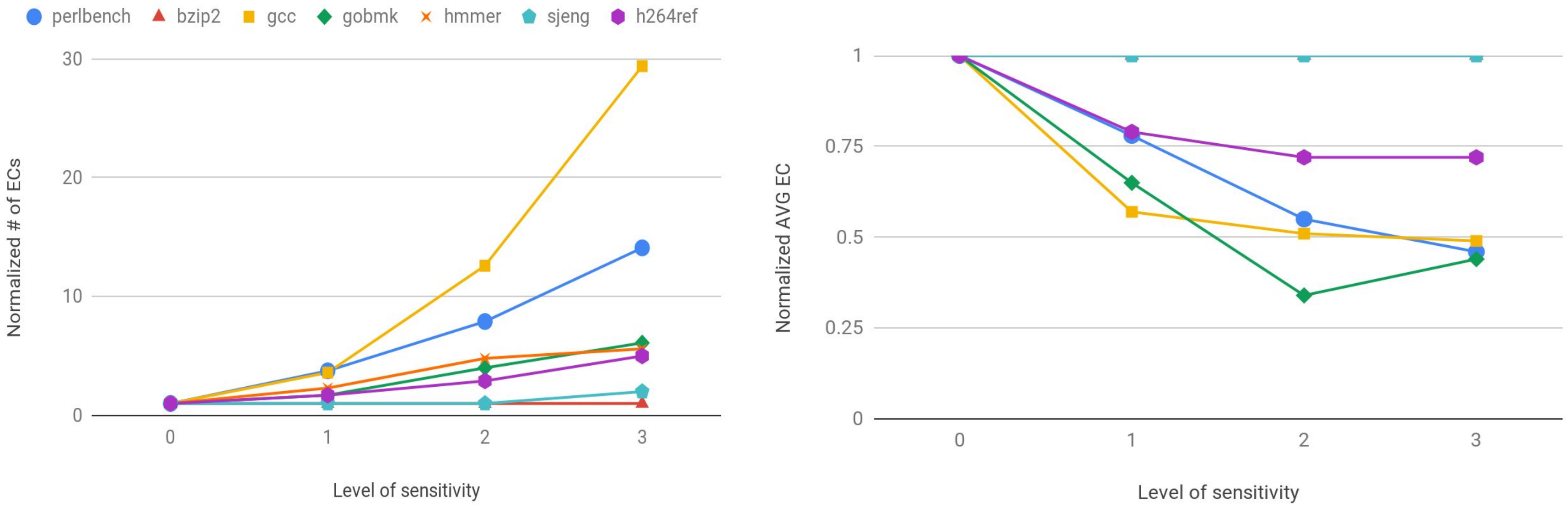
Evaluation

We separate our evaluation:

- Effectiveness
  - Call-site sensitivity
  - Localized concolic CFG
- Performance
- Exploitation

Call-site Sensitive as Context

Deeper context (call-site depth) provide better security.



Effectiveness of CFI-LB

Benchmark	Max Level	# of Indirect Calls	AVG_{EC}	Final AVG_{EC}	LC	$QS_{CFI\text{-}LB} / QS_{CFI(0)}$
400.perlbench	3	62/8/3/7	1.02/1.0/1.21/2.77	2.28	115	1/2.4
401.bzip2	0	12	1	1	1	1
403.gcc	2	161/23/36	1.48/1.37/1.50	1.47	54	1/1.84
445.gobmk	1	36/15/12	16.86/9.44/12.76	12.86	427	1/2.4
456.hmmer	0	9	1	1	1	1
464.h264ref	2	68/2/4/1	1.5/1.05/1.15/1.25	1.31	2	1/6.4
471.omnetpp	2	226/2/8/3	1.81/1.0/1.04/1.7	1.44	168	1/1.45
483.xalancbmk	2	1960/25/30/48	1.06/1.12/1.20/1.71	1.14	26	1/1.52
444.namd	0	12	1	1	1	1
447.dealII	2	100/3/4/1	1.04/1.0/1.0/1.11	1.03	2	1/1.07
450.soplex	0	56	1	1	1	1
453.povray	2	40/3/9	1.6/4.2/2.12	2.12	9	1/1.06
NGinx	3	94/18/0/11	5.54/1.06/0.0/4.91	3.73	62	1/3.3

TABLE III: Effectiveness of adaptive call-site sensitivity. The table shows the max call-site level, the number of indirect calls in each level, and for each level, the number of ECs and the average EC size. The last column shows the improvement of CFI-LB over context-insensitive CFI.

Effectiveness of Localized Concolic Execution

Benchmark	static-CFG	dyn-CFG	con-CFG	static-CFG\(dyn-CFG’ ∪ con-CFG’)	dyn-CFG \ con-CFG	con-CFG \ dyn-CFG
400.perlbench	879	1374	1387	41(4.66%)	0(0%)	13(0.94%)
401.bzip2	20	12	16	4(20%)	0(0%)	4(25%)
403.gcc	2198	3831	4125	94(4.28%)	14(0.37%)	308(7.47%)
445.gobmk	957	1882	1971	79(8.25%)	23(1.22%)	112(5.68%)
456.hmmer	52	47	59	6(11.54%)	0(0%)	12(20.34%)
458.sjeng	7	6	7	0(0%)	0(0%)	1(14.29%)
464.h264ref	711	262	479	206 (28.97%)	12(4.58%)	229(47.81%)

TABLE IV: Comparing static, dynamic, and concolic CFGs for the Spec CPU2006 benchmarks. Column 2 to 4 show the total number of entries in these CFGs, respectively. Note that number of static-CFG is not directly comparable to these of dyn-CFG and con-CFG because the latter two CFGs have contexts (hence more entries).

Performance Overhead

- Intel core-i7 6700 processor (skylake) with a base frequency of 3.4GHz and 16GB of memory
- Spec CPU2006 benchmark and NGinx
  - 2.7% for the forward-edge protection and 4.8% for the full protection.

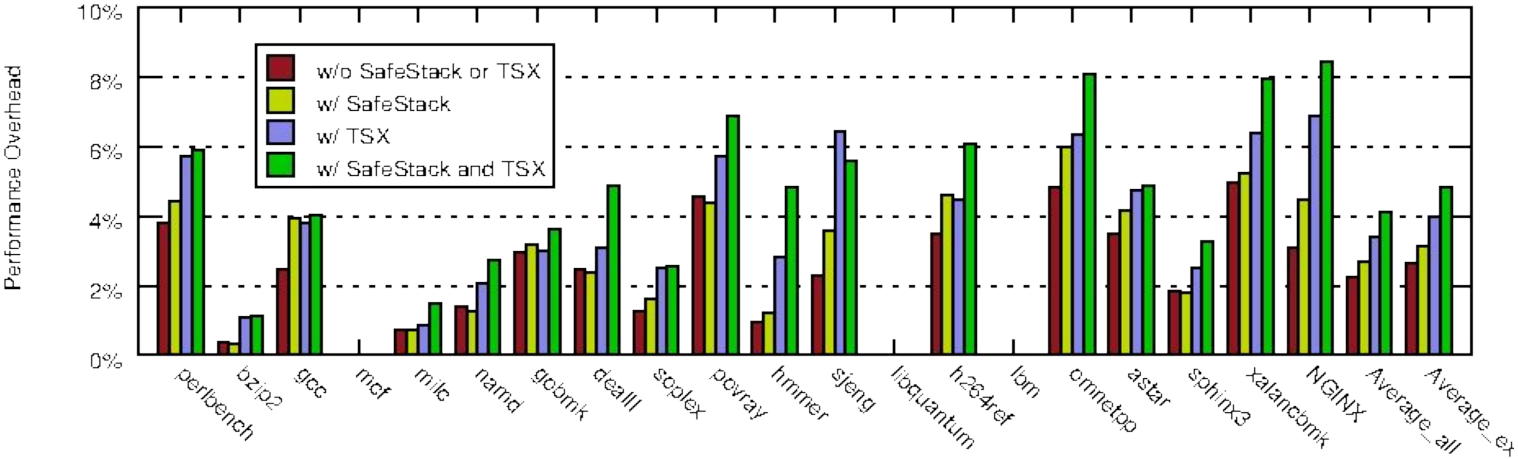
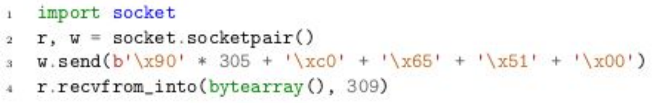


Fig: Performance overhead, Average_ex shows the average overhead excluding the three benchmarks that have no overhead.

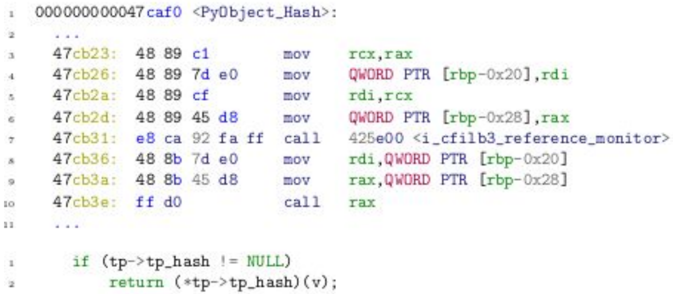
Exploitation Example

- CVE-2014-1912 (python-2.7.6)
- Root cause: missing check of the buffer size and the receive size in Python's socket module



```
import socket
r, w = socket.socketpair()
w.send(b'\x90' * 305 + '\xc0' + '\x65' + '\x51' + '\x00')
r.recvfrom_into(bytearray(), 309)
```

Fig. 8. PoC exploit for Python CVE-2014-1912



```
000000000047caf0 <PyObject_Hash>:
  ...
  47cb23:  48 89 c1         mov    rcx,rax
  47cb26:  48 89 7d e0      mov    QWORD PTR [rbp-0x20],rdi
  47cb2a:  48 89 cf         mov    rdi,rcx
  47cb2d:  48 89 45 d8      mov    QWORD PTR [rbp-0x28],rax
  47cb31:  e8 ca 92 fa ff   call   425e00 <i_cfilb3_reference_monitor>
  47cb36:  48 8b 7d e0      mov    rdi,QWORD PTR [rbp-0x20]
  47cb3a:  48 8b 45 d8      mov    rax,QWORD PTR [rbp-0x28]
  47cb3e:  ff d0            call   rax
  ...
```

```
    if (tp->tp_hash != NULL)
        return (*tp->tp_hash)(v);
```

Fig. 9. The execution of the hijacked function pointer

- CFI-LB at exploitable call-point:
  - Use call-site depth 3
  - Allowed 5 valid targets
  - Avg. EC size: 1.0
- CFI-LB can detect any control hijack at this point

Discussion

- Scalable context-sensitive static points-to analysis
  - Unavailable before mid 2018.
  - SUPA has open-sourced in july, 2018.
  - SUPA is based on SVF. It is both scalable and context-,flow- and field-sensitive.
  - Though, it fails to solve large number of pointers and returns imprecise points-to result.
- The offline log can be automated to verify and update CFG.
  - We keep it as a future work.

Benchmarks	Out of Budget			Empty Points-to Set	
	# of ICTs	SUPA	Type	# of ICTs	Type
400.perlbench	54	639	349	2	7
403.gcc	46	544	218	20	107
445.gobmk	22	1645	1637	1	4
447.dealII	0	-	-	23	37
450.soplex	0	-	-	157	11
471.omnetpp	37	143	44	67	21
483.xalancbmk	0	-	-	349	29

TABLE V: Failed cases of SUPA and the improvements of our type-based matching. Column 3, 4, and 6 show the largest EC sizes for SUPA and the type-based matching. SUPA works for all other benchmarks.

Conclusion

- Call-site sensitivity as context-sensitive CFI
- CFI-LB:
    - Fast and secure inline reference monitor
    - Complete and precise CFG
- Performance overhead:
    - 2.7% for the forward-edge protection
    - 4.8% for the full protection.
- Open-source: https://github.com/mustakcsecuet/CFI-LB

Q/A

Thank you

Protected Call Stack Solutions

A number of solutions are available and in progress:

- SafeStack
  - separates return addresses and others safe data into a separate safe stack.
  - published in OSDI’14 and adopted by LLVM in 2015 (still active in service, clang-9.0).
- ShadowCallStack
  - stores only the array of return addresses (contrast to SafeStack).
  - available for aarch64 in LLVM (from clang-7.0).
- Intel Control-flow Enforcement Technology (CET)
  - would add native support to use a shadow stack to store/check return addresses at call/return time.
  - would not suffer from race conditions and would not incur the overhead of function instrumentation.
  - recent update is on May 2019. (active research)

Path Sensitivity Pitfalls

Two most popular system use for runtime path tracing are:

- Intel Last Branch Record (LBR)
  - record the most recent 16/32 branches.
  - requires kernel modification and runtime access.
  - applied in PathArmor.
    - limited to paths before seven sensitive syscalls.
- Intel Processor Tracing (PT)
  - introduce for offline debug purpose
  - record whole execution path history but in compressed packets
  - requires kernel module for expensive decompression
  - applied in PittyPat, GRIFFIN, etc:
    - entire execution leading to the sensitive syscalls
  - also applied in μCFI:
    - reports data loss problem from Intel PT
    - for gcc, dealII, povray, omnetpp and xalancbmk spec benchmark

Static Points-to Analysis Pitfalls

A technical pitfalls on SVF/SUPA:

```
class A {
public:
  int f();
  int (A::*x)();
};

int A::f() { return 1; }

void ctx() {
  A a;
  a.x = &A::f;
  (a.*(a.x))();
}
```

```
##<> Source Loc:
   Ptr 4295                 PointsTo: {empty}
```

A conceptual pitfalls on SVF/SUPA:

```
fnptr instArr[] = {fn_add, fn_mul, fn_mov};
void buildInst(int i){
   instArr[i]();
}
void createAdd(){
   buildInst(0);
}
void createMov(){
   buildInst(2);
}
```

```
##<> Source Loc: (no context found)
   Ptr 4056             PointsTo: {fn_addr, fn_mul, fn_mov}
```

Improvements over Small Set Seeds

Benchmark	dyn-CFG-r	dyn-CFG-t	con-CFG-t	dyn-CFG-r \ dyn-CFG-t	dyn-CFG-r \ con-CFG-t	Discovered
400.perlbench	1374	449	1051	925	323(23.51%)	602
401.bzip2	12	12	16	0	0(0%)	0
403.gcc	3831	2196	3929	1635	53(1.38%)	1582
445.gobmk	1882	1102	1833	780	49(2.60%)	731
456.hmmer	47	3	58	44	1(2.13%)	43
458.sjeng	6	6	7	0	0(0%)	0
464.h2564ref	262	240	473	220	12(4.58%)	4

TABLE VI: Comparing concolic and dynamic CFGs. dyn/con-CFG-t is derived from the small test inputs; dyn/con-CFG-r is derived from the large reference inputs. Our localized concolic execution can discover most of the control transfers in the dyn-CFG-r using only the small inputs.