Origin-sensitive CFI

Mustakimur R. Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, Jie Yang

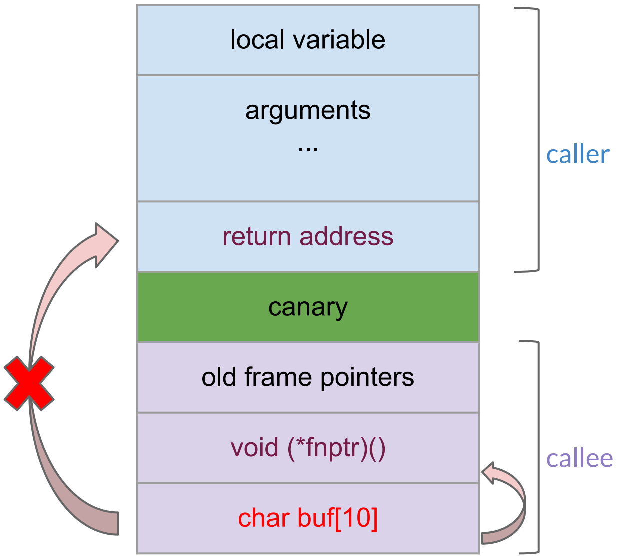
Department of Computer Science
Florida State University

Control Flow Integrity

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

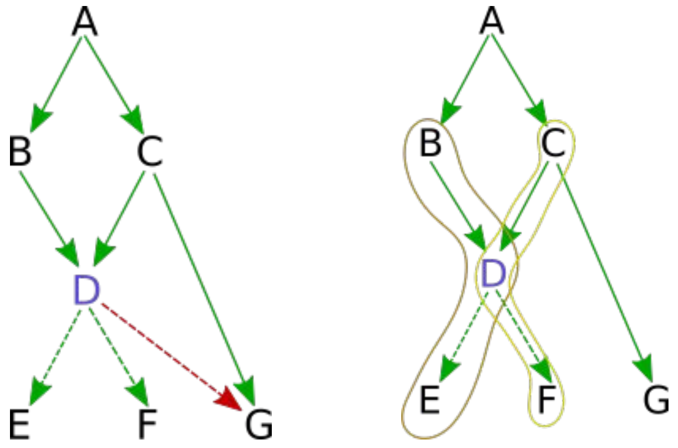
CFI consists of:

- CFI Policy
- Inline Reference Monitor
- CFG



Context-sensitive 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, call-site sensitivity



To quantify security guarantee of CFI:

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

Some Context-Sensitive CFI systems cannot break down largest ECs (limited number of contexts, i.e., incoming execution paths to ICTs)

Motivation

- There is a C-style indirect call from *execute()*.
  - That function pointer, *code_to_exec* is a member of an object.
- The context of the indirect call is a loop that iterates over a list of objects.
  - Indirect calls has **indifferent context**.
- The function pointer receives the target when the object is created.
  - The objects are created from different locations.
- The object creation location is more diverse than the context of the indirect call.
  - Object creation location aka origin.

```
#define EXECUTE_ON_STARTUP(NAME, CODE)  \
  static void __##NAME##_code() {CODE;} \
  static ExecuteOnStartup __##NAME##_reg(__##NAME##_code);

#define Define_Network(NAME) \
  EXECUTE_ON_STARTUP(NAME##__net,\
  (new NAME(#NAME))->setOwner(&networks);)

Define_Network(smallLAN);
Define_Network(largeLAN);

class ExecuteOnStartup{
private:
  void (*code_to_exec)();
  ExecuteOnStartup *next;
  static ExecuteOnStartup *head;
public:
  ExecuteOnStartup(void (*_code_to_exec)()){
    code_to_exec = _code_to_exec;
    // add to list
    next = head;
    head = this;
  }
  void execute(){
    code_to_exec();
  }
  static void executeAll(){
    ExecuteOnStartup *p = ExecuteOnStartup::head;
    while (p){
      p->execute();
      p = p->next;
    }
  }
};
void cEnvir::setup(...){
  try{
    ExecuteOnStartup::executeAll();
  }
}
```

Largest EC Size Case Study from 471.omnetpp benchmark

Origin Sensitivity: A New Type of Context

- **Origin**: code location where a code pointer originates.
  - **Virtual call:** where the receiving object is created (**class constructor is being called**).
  - **C-style indirect call:** the address-taken code location of the code pointer.
- Requires an efficient run-time tracing method.
  - Map object's virtual pointer to object construction location.
  - Map code pointer to address-taken location.
- Performance is a challenge:
  - Track origins as function addresses propagate throughout the program
  - Similar to how taint is tracked.

```
typedef void (*fnptr)();
void target(){
}
void callee(fnptr arg){
  fnptr tmp = arg;
  tmp();
}
void caller(){
  callee(&target);
}
```

Hybrid Definition

- Need a more efficient definition for C-style ICT.
  - Combines the origin with call-site sensitivity.
  - **Origin: latest code pointer assignment location.**
  - **Use call-sites as the context for the origin.**
- Virtual function does not need change
  - Constructors cannot be virtually called
  - If an object is copied to another object, it essentially create a new object using its class' copy constructor or copy assignment operator. This creates a new origin for that object.

```
typedef void (*fnptr)();
void target(){
}
void callee(fnptr arg){
  fnptr tmp = arg;
  tmp();
}
void caller(){
  callee(&target);
}
```

origin

call-site

Origin Sensitivity Effectiveness

- As compared to call-site sensitivity

Benchmarks	Lang	Context-insensitive	1-call-site		2-call-site		origin-sensitive	
		EC_L	EC_L	Reduce by	EC_L	Reduce by	EC_L	Reduce by
445.gobmk	C	427	427	0%	427	0%	427	0%
400.perlbench	C	173	120	31%	113	35%	21	88%
403.gcc	C	54	54	0%	54	0%	42	22%
471.omnetpp	C++	168	168	0%	168	0%	2	99%
483.xalancbmk	C++	38	38	0%	38	0%	4	95%
453.povray	C++	11	11	0%	11	0%	10	10%

445.gobmk: because it contains a loop over a large static array of function pointers (the owl_defendpat array).

OS-CFI

- LLVM-based prototype OS-CFI system.
- Focus on:
    - **Precision:** OS-CFI must improve the security by reducing the average and largest EC sizes.
    - **Security:** OS-CFI must protect both the contextual data and the (temporary) data used by reference monitors.
    - **Performance:** OS-CFI must have strong performance relative to the native system.
    - **Compatibility:** OS-CFI must support both C and C++ programs.

OS-CFI Policy

- Adaptive CFI policy:
    - Use call-site sensitivity if it is sufficiently precise
    - Use origin sensitivity to break down large ECs

Instrumentation

```
typedef void (*Format)();
class Base {
protected:
  Format fmt;
public:
  Base(/* Base_o.vPtr, origin */) {
    // store_metadata(Base_o.vPtr, Base::vTavle,
    //                origin);
  }
  ~Base() {}
  virtual void set(Format fp) {
    fmt = fp;
    // store_metadata(fmt.addr, fp.value,
    //                Base:set_loc1, Base::set_ctx);
  }
  void print() {
    // ccall_ref_monitor(fmt.addr, fmt.value);
    fmt();
  }
};
```

```
class Child : public Base {
public:
  Child(/* Child_o.vPtr, origin */) {
    // Base(Child_o.vPtr, origin);
    // store_metadata(Child_o.vPtr, Child::vTable,
    //                origin);
  }
  ~Child() {}
  void set(Format fp) {
    fmt = fp;
    // store_metadata(fmt.addr, fp.value,
    //                Child::set_loc1, Child::set_ctx);
  }
  void print() {
    // ccall_ref_monitor(fmt.addr, fmt.value);
    fmt();
  }
};
```

```
void exec () {
  Base *bp = new Base(); // call constructor
  // vcall_ref_monitor(Base_o.vPtr,
  //                   Base::vTable, Base::set())
  bp->set(&targetA);
  bp->print();

  Child ci;              // call constructor
  ci.set(&targetB);
  ci.print();

  bp = &ci;
  // vcall_ref_monitor(Child_o.vPtr,
  //                   Child::vTable, Child::set())
  bp->set(&targetB);
  bp->print();
}
```

- To track origin of the object creation location.
  - store_metadata(vptr_addr, vtable, origin_loc)
- To track origin of the function pointer assignment location.
  - store_metadata(ptr_addr, ptr_val, origin_loc, origin_context)
- To monitor the virtual function call.
  - ccall_ref_monitor(ptr_addr, target)
- To monitor the C-style indirect call.
  - vcall_ref_monitor(vptr_addr, vtable, target)

CFG Generation

- Based on SUPA, an on-demand context-, flow-, and field-sensitive points-to analysis
  - Constructs a whole-program sparse value-flow graph (SVFG) that conservatively captures the program's (imprecise) **def-use chains**.
  - Improves the precision by refining away imprecise value-flows in the SVFG with strong updates.
- OS-CFI CFGs are constructed on top of the refined SVFG of SUPA.
  - Piggybacks on SUPA while **traversing the program's SVFG** reversely to compute points-to sets.
  - **Reverse: from sink (ICT) to source (origin)**.

Pitfalls (Static Points-to Analysis)

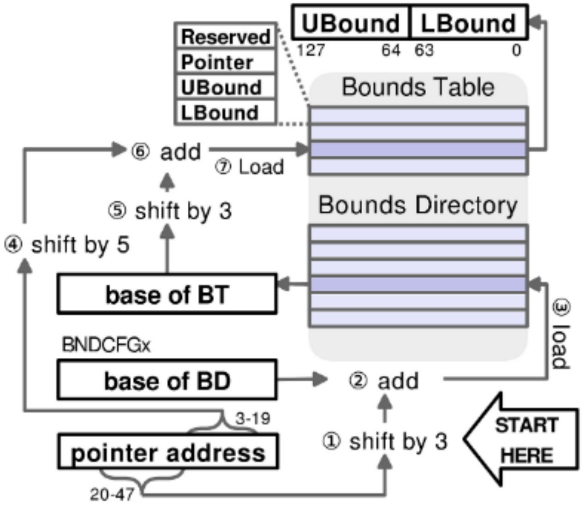
- SUPA is Scalable, precise, and publicly available.
  - Relatively powerful machine (16-core Xeon server with 64GB of memory).
- Issues
  - Out of budget
    - Generous budget (-maxcxt=10 -flowbg=10000 -cxtbg=100000).
    - Returns set of address-taken functions (refined by including type-matched).
  - Empty points-to sets
    - **Mostly because of missing implementations** e.g. pointer to member function.
    - Refined by address-taken and type-matched set.

Benchmarks	Out of budget			Empty points-to sets	
	# 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
453.porvray	47	317	79	22	24
471.omnetpp	37	143	44	67	21
483.xalancbmk	0	-	-	349	29
NGINX	141	1066	102	4	34

Table 2: 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.

Metadata Storage

- Intel MPX is a Hardware-based bound check system.
  - Operates like a two-level page table.
- Repurpose MPX as **a generic (key, value) store**
  - Indexed by the address of a pointer (code pointer address).
  - Every bound table entry consists of
    - content of the pointer (code pointer target).
    - the upper bound (origin location).
    - the lower bound (origin context).
  - Map (ptr_addr, ptr_target) = <origin, origin_context>.
  - If inline reference monitor
    - Provide wrong ptr_target, load will fail.
    - Provide correct ptr_target, origin and origin context will be



Protection of Metadata, Context, IRM

- Intel MPX (Runtime Metadata)
  - Protected by ASLR.
  - Bound directory (user-space), Bound Table (kernel space).
  - **Base of the bounds directory is stored in a special register, BNDCFGx, inaccessible to the user space.**
  - With additional overhead, MPX's bound check can be used to protect itself.
- Context (Call Stack)
  - Intel CET shadow stack (recent update is on May 2019.
  - SafeStack (published in OSDI'14 and adopted by LLVM in 2015 (clang-9.0)).
  - ShadowCallStack (available for aarch64 in LLVM (from clang-7.0)).
- Reference Monitor protected by Intel TSX
  - keeps tracks of the memory accessed by a transaction and aborts the transaction if any of that memory is changed by others.

CFG Address Mapping

- CFGs are accordingly encoded as the LLVM IR locations.
  - But runtime Requires the low-level addresses of the CFG nodes.
- Traditional approach
  - Use the debug information
    - works for function addresses.
    - but not as well for call sites because they are not in the symbol table.
  - Use heuristics
    - such as the code structure are used to infer the locations of call sites.
    - may not be reliable when the compiler optimization is turned on.
- OS-CFI uses **Label-As-Value** to obtain the runtime addresses of the CFG nodes
  - Create a label at every required call-sites
  - Create an array of label in required functions and located it into a custom section
  - Assembler will automatically convert the label with actual code address
  - **Supports ASLR**

Evaluation

We separate our evaluation into three parts:

- Improvements in security
  - Security guarantee
  - Case study
- Experiments with vulnerabilities
- Performance

Security Guarantee (1)

- Excluded SUPA failed cases.
- Comparing to CI-CFI
  - Average Avg. EC Size reduction 59.8%.
  - Average Largest EC Size reduction 60.2%.

Benchmark	# ICTs	No Context		OS-CFI		Reduce by	
		Avg	Lg	Avg	Lg	Avg	Lg
400.perlbench	79	23.8	39	2.8	10	88%	74%
401.bzip2	20	2.0	2	1.0	1	50%	50%
403.gcc	347	30.7	169	1.3	27	96%	84%
433.milc	4	2.0	2	1.0	1	50%	50%
445.gobmk	36	8.1	107	1.5	12	82%	89%
456.hmmer	9	2.8	10	1.0	1	64%	90%
464.h264ref	367	2.0	12	1.0	2	50%	83%
444.namd	12	2.5	3	1.0	1	60%	67%
447.dealII	79	2.1	3	1.2	3	43%	0%
450.soplex	317	1.0	1	1.0	1	0%	0%
453.porvray	45	9.3	17	1.6	5	83%	71%
471.omnetpp	331	5.7	109	1.0	2	83%	98%
473.astar	1	1.0	1	1.0	1	0%	0%
483.xalancbmk	1492	2.5	11	1.0	1	60%	91%
NGINX	248	9.4	43	1.1	19	88%	56%

Table 3: Improvement of precision by OS-CFI over context-insensitive CFI, shown by the significant reduction in the average (`Avg`) and largest (`Lg`) EC sizes.

Security Guarantee (2)

Benchmark	#ICTs		OS-CFI / Adaptiveness												
			Origin sensitive				Call-site sensitive				Context-insensitive			Overall	
	#c-Call	#vCall	#ICTs	#Origins	Avg	Lg	#ICTs	Depth	Avg	Lg	#ICTs	Avg	Lg	Avg	Lg
400.perlbench	135	0	53	49	2.5	6	18	2	3.2	8	64	25.5	349	11.4	349
401.bzip2	20	0	20	4	1.0	1	0	0	0	0	0	0	0	1.0	1
403.gcc	413	0	249	139	1.0	1	88	2	1.0	1	76	29.8	218	3.4	218
433.milc	4	0	0	0	0	0	4	1	1.0	1	0	0	1	1.0	1
445.gobmk	59	0	29	12	1.4	3	7	3	1.0	1	23	661.7	1637	246.3	1637
456.hmmer	9	0	1	15	1.0	1	1	1	1.0	1	7	1.0	1	1.0	1
458.sjeng	1	0	0	0	0	0	0	0	0	0	1	7	7	7.0	7
464.h264ref	367	0	318	52	1.0	1	7	1	1.5	2	42	1.7	2	1.1	2
444.namd	12	0	12	30	1.0	1	0	0	0	0	0	0	0	1.0	1
447.dealII	7	95	73	59	1.0	1	3	2	1.0	1	26	27.9	37	6.7	37
450.soplex	0	357	0	0	0	0	0	0	0	0	357	1.2	11	1.2	11
453.porvray	38	76	37	29	1.5	5	8	3	1.0	1	69	14.4	79	7.5	79
471.omnetpp	39	403	276	243	1.0	1	21	2	1.0	1	145	27.5	44	9.2	44
473.astar	0	1	0	0	0	0	0	0	0	0	1	1.0	1	1.0	1
483.xalancbmk	18	2073	1486	1544	1.0	1	6	3	1.0	1	599	7.2	29	3.5	29
NGINX	393	0	184	169	1.0	1	37	3	1.0	1	172	13.8	102	6.6	102

Table 4: Overall distribution of ICTs among origin sensitive, call-site sensitive, and context-insensitive ICTs. The second column shows the total number of C-style indirect calls, while the third column shows the number of virtual calls. We omit the results of `mcf`, `libquantum`, and `sphinx3` from this table because they do not have ICTs in their main programs.Columns marked with `Avg` and `Lg` show the average and largest EC sizes, respectively.

Case Study

```
class cObject{
protected:
  void discard(cObject *object){
    if(object->storage() == 'D')
      delete object;
    else
      object->setOwner(NULL);
  }
public:
  virtual ~cObject();
}
class cModuleType:public cObject{
  ~cModule(){
    delete [] fullname;
  }
}
class cArray:public cObject{
private:
  cObject **vect;
public:
  clear(){
    for (int i=0; i<=last; i++){
      if (vect[i] && vect[i]->owner()==this)
        discard( vect[i] );
    }
  }
}
```

Figure 5: Virtual call with the largest EC in `471.omnetpp`

```
class XMLRegisterCleanup
{
private:
  XMLCleanupFn m_cleanupFn;
public :
  void registerCleanup(XMLCleanupFn cleanupFn) {
    m_cleanupFn = cleanupFn;
  }
  void doCleanup() {
    if (m_cleanupFn)
      m_cleanupFn();
  }
}
XMLTransService::XMLTransService(){
  static XMLRegisterCleanup mappingsCleanup;
  static XMLRegisterCleanup mappingsRecognizerCleanup;

  mappingsCleanup.registerCleanup(reinitMappings);
  mappingsRecognizerCleanup.registerCleanup
                          (reinitMappingsRecognizer);
}
```

Figure 6: The ICT with the largest EC in `483.xalancbmk`

Pitfalls (CFI Policy)

- This single ICT can target to 8 functions.
    - The target is decided by the index *piecet(i)*.
- SUPA fails to provide the context for the ICT.
    - Because *evalRoutines* is **initialized statically**, SUPA will not generate any context for this ICT.
- This case requires **to protect the integrity of index data** throughout its context.

```
1  typedef int (*EVALFUNC)(int sq,int c);
2  static EVALFUNC evalRoutines[7] = {
3    ErrorIt,
4    Pawn,
5    Knight,
6    King,
7    Rook,
8    Queen,
9    Bishop };
10
11 int std_eval (int alpha, int beta) {
12   for (j = 1, a = 1; (a <= piece_count); j++) {
13     score += (*(evalRoutines[piecet(i)]))
14                             (i,pieceside(i));
15   }
16 }
```

Figure 4: An example in `sjeng` where the ICT at Line 15 has no context in SUPA.

Synthesized Exploit

- Background
  - Two virtual function calls.
  - Two vulnerable functions
    - *getPerson()* may return a malicious object by overwriting the *vPtr* with wrong *vTable*.
    - *isEmployee()* may always return *true* by overwriting boolean return.
- Security guarantee
  - First ICT is protected by Object Type Integrity.
  - Second ICT is protected by CFI.

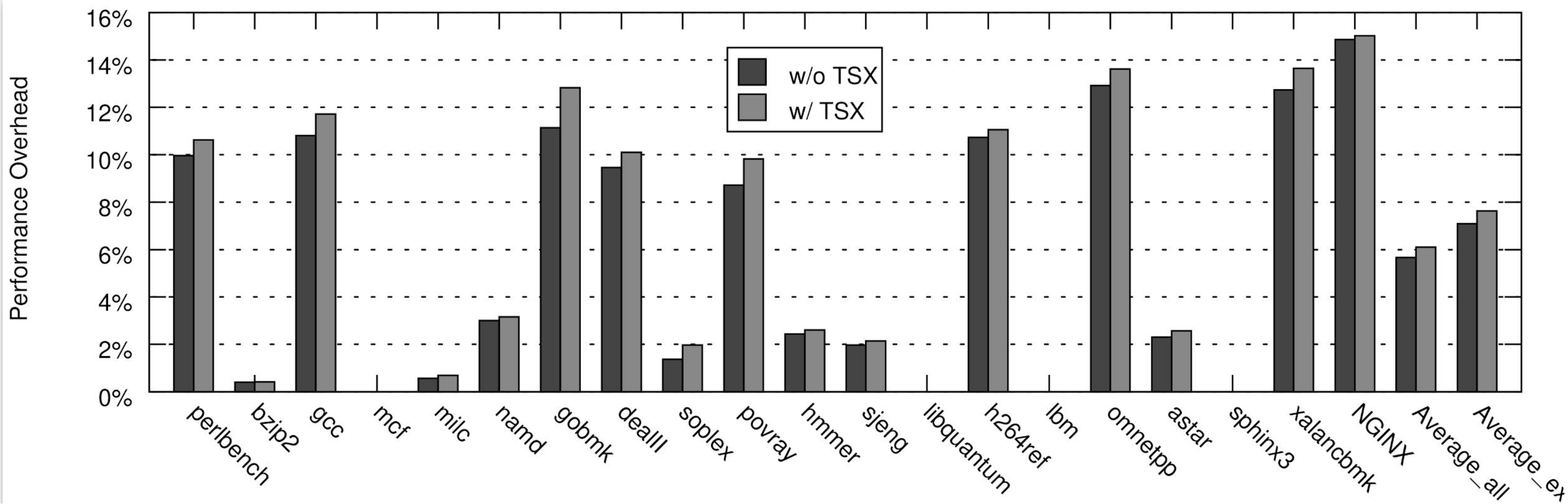
```
class Person{
protected:
  SalaryAccount *salary = nullptr;
public:
  virtual void seeEvaluation()=0;
  virtual void seeSalary(){/*null derefernce*/}
};
class Employee : public Person{
public:
  void seeEvaluation(){/* show employee evaluation*/}
  void seeSalary(){/*employee has salary account*/}
};
class Employer : public Person{
public:
  void seeEvaluation(void){/*list of employee evaluation*/}
};
Person *getPerson(int id){
  char *name = (char*)malloc(10);
  Person *p;
  if (isEmployer(id)) {
    p = new Employer();
  } else {
    p = new Employee();
  }
  gets(name); // vulnerable gets()
  ...
  return p;
}
bool isEmployee(Person *member) {
  bool res = false;
  char name[10];
  // vulnerable strcpy()
  strcpy(name, member->getName());
  ...
  if(Employee *emp = dyn_cast<Employee*>(member)){
    if(emp != NULL)
      res = true;
  }
  return res; // attacker overwrite res
}
int main() {
  ...
  Person *member;
  member = getPerson(id);

  // if employee, can only see his/her evaluation
  // if employer, can see list of employee evaluation
  member->seeEvaluation(); // OTI protected

  // only employee has salary account
  if (isEmployee(member))
    member->seeSalary(); // CFI protected
}
```

Figure 10: A program vulnerable to COOP attack.

Performance



- Intel Xeon E3-1275 processor and 64 GB of memory.
- **SafeStack** for secure call stack and **Intel TSX** to protect the reference monitors.
- OS-CFI incurred an overhead of **7.1%** without Intel TSX and **7.6%** with it.
- CFG generation has no longer than **5.3%** overhead.

Related Work

Categories	CFIXX	PathArmor	PittyPat	μCFI	OS-CFI
Protected	Object type	Control flow	Control flow	Control flow	Control flow & Object type
Context	`vPtr` to `vTable` binding	last branches taken	Processor execution paths	Execution paths and constraint data	Origins of function pointers and objects
CFG	None	On-demand, constraint driven context-sensitive CFG	Abstract-interpretation based online points-to analysis	Run-time points-to analysis	CFGs based on context-, flow- and field-sensitive static points-to analysis
Coverage	Virtual calls	Selected syscalls	Whole program, enforced at selected syscalls	Whole program, enforced at selected syscalls	Whole program, enforced at every ICT
Required hardware	Intel MPX for metadata storage	Intel LBR for taken branches	Intel PT for execution history	Intel PT for execution history and control data	Intel MPX for metadata storage and Intel TSX to protect reference monitors
Kernel changes	No, built-in MPX support	Yes, enforce CFI on the syscall boundary	Yes, redirect traces and enforce CFI on syscall boundary	Yes, redirect traces and enforce CFI on syscall boundary	No, built-in MPX and TSX support
Runtime support	Library to track the type of each object	Per-thread control transfer monitoring	Additional threads to parse trace and verify control flow	Additional threads to parse trace and verify control flow	Hash based verification protected by TSX

Table 6: Comparison between OS-CFI and recent (context-sensitive) CFI systems

- CPI is another closely related work, it protected the integrity of all the code pointers

Conclusion

- **Origin sensitivity** is an **effective context** for CFI to **reduce the LC size**.
- OS-CFI supports both **virtual calls and C-style ICTs.**
- Repurposing **Intel MPX as generic (key, value) store.**
- **Static points-to analysis** for CFG generation **requires special attention** to ensure the security guarantee.
- Source code available: https://github.com/mustakcsecuet/OS-CFI

Q&A

https://github.com/mustakcsecuet/OS-CFI

Performance of CFG Generator

Benchmark	SUPA (s)	OS-CFI (s)	Overhead
400.perlbench	6083.2	6350.7	4.4%
401.bzip2	445.8	457.2	2.6%
403.gcc	53029.1	56231.7	6.0%
433.milc	3.9	4.0	2.6%
445.gobmk	4071.5	4246.4	4.3%
456.hmmer	10.9	11.8	8.3%
458.sjeng	2.6	2.6	0.0%
464.h264ref	372.1	382.0	2.7%
444.namd	15.6	16.7	7.1%
447.dealII	651.5	673.8	3.5%
450.soplex	1280.7	1340.2	4.6%
453.povray	4633.9	5304.0	14.5%
471.omnetpp	43929.0	45351.5	3.2%
473.astar	1.4	1.5	7.1%
483.xalancbmk	9703.7	10792.6	11.2%
NGINX	39860.2	41630.7	4.4%
Average	10255.9	10799.8	5.3%

Table 5: The analysis time of OS-CFI as compared to the vanilla SUPA algorithm. The unit of the analysis time in the table is seconds.

Real-world Exploit

- Based on **CVE-2015-8668**
  - Heap-based buffer overflow caused by an integer overflow.
- Overwrite *TIFF* object *out* using the overflow vulnerability.

```c
int TIFFWriteScanline(TIFF* tif, ...){
  ...
  status = (*tif->tif_encoderow)(tif, (uint8*) buf,
    tif->tif_scanlinesize, sample); // <= exploit call-point
}
void _TIFFSetDefaultCompressionState(TIFF* tif){
  tif->tif_encoderow = _TIFFNoRowEncode; // <= origin
}
TIFF* TIFFOpen(...){
  ...
  _TIFFSetDefaultCompressionState(tif);
}
int main(int argc, char* argv[]){
  TIFF *out = NULL;
  out = TIFFOpen(outfilename, "w"); // <= exploited object
  ...
  uint32 uncompr_size;
  unsigned char *uncomprbuf;
  ...
  uncompr_size = width * length; // non-sanitized code and
                                 // following memory allocation
  uncomprbuf = (unsigned char *)_TIFFmalloc(uncompr_size);
  ...
  if (TIFFWriteScanline(out, ...) < 0) {}
   ...
}
```

Figure 8: Sketch of the vulnerable code in libtiff v4.0.6.