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:

$$QSCFI = \text{AVG}_{EC} \times LC$$

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.

```c
#define EXECUTE_ON_STARTUP(NAME, CODE) \
    static void ___NAME##_code() {CODE;} \
    static ExecuteOnStartup ___NAME__reg(___NAME##_code);
#define Define_Network(NAME) \n    EXECUTE_ON_STARTUP(NAME##_net,\n    (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; 
            } 
        }
    };
    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.

```c
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.

```c
typedef void (*fnptr)();
void target()
{
}
void callee(fnptr arg)
{
    fnptr tmp = arg;
    tmp();
}
void caller()
{
    callee(&target);
}
```
# Origin Sensitivity Effectiveness

- As compared to call-site sensitivity

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Lang</th>
<th>Context-insensitive</th>
<th>1-call-site</th>
<th>2-call-site</th>
<th>origin-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>EC&lt;sub&gt;L&lt;/sub&gt;</td>
<td>EC&lt;sub&gt;L&lt;/sub&gt;</td>
<td>Reduce by</td>
<td>EC&lt;sub&gt;L&lt;/sub&gt;</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>C</td>
<td>427</td>
<td>427</td>
<td>0%</td>
<td>427</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>C</td>
<td>173</td>
<td>120</td>
<td>31%</td>
<td>113</td>
</tr>
<tr>
<td>403.gcc</td>
<td>C</td>
<td>54</td>
<td>54</td>
<td>0%</td>
<td>54</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>C++</td>
<td>168</td>
<td>168</td>
<td>0%</td>
<td>168</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>C++</td>
<td>38</td>
<td>38</td>
<td>0%</td>
<td>38</td>
</tr>
<tr>
<td>453.povray</td>
<td>C++</td>
<td>11</td>
<td>11</td>
<td>0%</td>
<td>11</td>
</tr>
</tbody>
</table>

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.
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)`

```c
typedef void (*Format)();
class Base {
  protected:
    Format fmt;
  public:
    Base(/* Base.o.vPtr, origin */) {
      // store_metadata(Base.o.vPtr, Base::vTable, // origin);
    }
    virtual void set(Format fp) {
      fmt = fp;
      // store_metadata(fmt.addr, fp.value, // Base::set_loc1, Base::set_ctz);
    }
    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_ctz);
    }
    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();
}
```
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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Out of budget</th>
<th>Empty points-to sets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of ICTs</td>
<td>SUPA</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>54</td>
<td>639</td>
</tr>
<tr>
<td>403.gcc</td>
<td>46</td>
<td>544</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>22</td>
<td>1645</td>
</tr>
<tr>
<td>447.dealII</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>450.soplex</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>453.porray</td>
<td>47</td>
<td>317</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>37</td>
<td>143</td>
</tr>
<tr>
<td>483.xalanchmk</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>NGINX</td>
<td>141</td>
<td>1066</td>
</tr>
</tbody>
</table>

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 verified.
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%.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># ICTs</th>
<th>No Context</th>
<th>OS-CFI</th>
<th>Reduce by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Avg</td>
<td>Lg</td>
<td>Avg</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>79</td>
<td>23.8</td>
<td>39</td>
<td>2.8</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>20</td>
<td>2.0</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>403.gcc</td>
<td>347</td>
<td>30.7</td>
<td>169</td>
<td>1.3</td>
</tr>
<tr>
<td>433.mile</td>
<td>4</td>
<td>2.0</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>36</td>
<td>8.1</td>
<td>107</td>
<td>1.5</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>9</td>
<td>2.8</td>
<td>10</td>
<td>1.0</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>367</td>
<td>2.0</td>
<td>12</td>
<td>1.0</td>
</tr>
<tr>
<td>444.namd</td>
<td>12</td>
<td>2.5</td>
<td>3</td>
<td>1.0</td>
</tr>
<tr>
<td>447.dealII</td>
<td>79</td>
<td>2.1</td>
<td>3</td>
<td>1.2</td>
</tr>
<tr>
<td>450.soplex</td>
<td>317</td>
<td>1.0</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>453.porray</td>
<td>45</td>
<td>9.3</td>
<td>17</td>
<td>1.6</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>331</td>
<td>5.7</td>
<td>109</td>
<td>1.0</td>
</tr>
<tr>
<td>473.astar</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>483.xalancBmk</td>
<td>1492</td>
<td>2.5</td>
<td>11</td>
<td>1.0</td>
</tr>
<tr>
<td>NGINX</td>
<td>248</td>
<td>9.4</td>
<td>43</td>
<td>1.1</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#ICTs</th>
<th>Origin sensitive</th>
<th>Call-site sensitive</th>
<th>Context-insensitive</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#c-Call</td>
<td>#vCall</td>
<td>#ICTs</td>
<td>#Origins</td>
<td>Avg</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>135</td>
<td>0</td>
<td>53</td>
<td>49</td>
<td>2.5</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>4</td>
<td>1.0</td>
</tr>
<tr>
<td>403.gcc</td>
<td>413</td>
<td>0</td>
<td>249</td>
<td>139</td>
<td>1.0</td>
</tr>
<tr>
<td>433.milc</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>59</td>
<td>0</td>
<td>29</td>
<td>12</td>
<td>1.4</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>15</td>
<td>1.0</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>367</td>
<td>0</td>
<td>318</td>
<td>52</td>
<td>1.0</td>
</tr>
<tr>
<td>444.namd</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td>30</td>
<td>1.0</td>
</tr>
<tr>
<td>447.deallII</td>
<td>7</td>
<td>95</td>
<td>73</td>
<td>59</td>
<td>1.0</td>
</tr>
<tr>
<td>450.soplex</td>
<td>0</td>
<td>357</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>453.porvray</td>
<td>38</td>
<td>76</td>
<td>37</td>
<td>29</td>
<td>1.5</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>39</td>
<td>403</td>
<td>276</td>
<td>243</td>
<td>1.0</td>
</tr>
<tr>
<td>473.astar</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>18</td>
<td>2073</td>
<td>1486</td>
<td>1544</td>
<td>1.0</td>
</tr>
<tr>
<td>NGINX</td>
<td>393</td>
<td>0</td>
<td>184</td>
<td>169</td>
<td>1.0</td>
</tr>
</tbody>
</table>

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 $\text{piecet}(i)$.
- 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.

```
typedef int (*EVALFUNC)(int sq, int c);
static EVALFUNC evalRoutines[] = {
    ErrorIt,
    Pawn,
    Knight,
    King,
    Rook,
    Queen,
    Bishop
};

int std_eval (int alpha, int beta) {
    for (j = 1, a = 1; (a <= piece_count); j++) {
        score += (*evalRoutines[piecet(i)]
                   (i, pieceside(i));
    }
}
```

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.

```c++
class Person{
  protected:
    SalaryAccount *salary = nullptr;
  public:
    virtual void seeEvaluation() = 0;
    virtual void seeSalary(){/*null dereference*/}
  }
};

class Employee : public Person{
  public:
    void seeEvaluation() {/* show employer evaluation*/
    void seeSalary(){/*employee has salary account*/
  }
};

class Employer : public Person{
  public:
    void seeEvaluation(void) {/*test of employee evaluation*/
}
};

Person *getPerson(int id){
    char *name = (char*)malloc(10);
    Person *p;
    if (!isEmployer(id)) {
      p = new Employer();
    } else {
      p = new Employee();
    }
    p->getName(); // vulnerable getName
    ...
    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->isNull())
        res = true;
    return res; // attacker overwrite res
}

int main() {
    ...
    Person *member;
    member = getPerson(id); // only employee can see his/her evaluation
    if (isEmployer())
      seeEvaluation(); // OTI protected
    if (isEmployee(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.
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](https://github.com/mustakcsecuet/OS-CFI)
Q&A

http://ww2.cs.fsu.edu/~khandake/
### Performance of CFG Generator

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

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.