Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis

Kangjie Lu

Hong Hu

University of Minnesota

Georgia Tech
What is an indirect call?
Example, purpose, and commonness

```c
void foo(int a) {
    printf("a = %d\n", a);
}

typedef void (*fptr_t)(int);

// Take the address of foo() and
// assign to function pointer fptr
fptr_t fptr = &foo;

...  

// Indirect call to foo()
fptr(10);
```
Example, purpose, and commonness

```c
void foo(int a) {
    printf("a = %d\n", a);
}
typedef void (*fptr_t)(int);

// Take the address of foo() and
// assign to function pointer fptr
fptr_t fptr = &foo;
...

// Indirect call to foo()
fptr(10);
```
Example, purpose, and commonness

```c
void foo(int a) {
    printf("a = %d\n", a);
}
typedef void (*fptr_t)(int);

// Take the address of foo() and
// assign to function pointer fptr
fptr_t fptr = &foo;

... // Indirect call to foo()
fptr(10);
```

- **Purpose**
  - To support dynamic behaviors

- **Common scenarios**
  - Interface functions
  - Virtual functions
  - Callbacks

- **Commonness**
  - Linux: 58K
  - Firefox: 37K
void foo(int a) {
    printf("a = %d\n", a);
}
typedef void (*fptr_t)(int);

// Take the address of foo() and
// assign to function pointer fptr
fptr_t fptr = &foo;
...

// Indirect call to foo()
fpTR(10);

- **Purpose**
  - To support dynamic behaviors

- **Common scenarios**
  - Interface functions
  - Virtual functions
  - Callbacks

Indirect calls are essential and common

- **Firefox**: 37K
Indirect call is however a major roadblock in security

Couldn’t construct a precise call-graph!
Indirect call is however a major roadblock in security

Couldn’t construct a precise call-graph!

- All inter-procedural static analyses and bug detection require a global call-graph!
  - Otherwise, path explosion and inaccuracy

- Effectiveness of control-flow integrity (CFI) depends on it!
Indirect call is however a major roadblock in security

- All inter-procedural static analyses and bug detection require a global call-graph!
  - Otherwise, path explosion and inaccuracy

Identifying indirect-call targets is foundational to security!
How can we identify them?
Two approaches: Point-to analysis vs. Type analysis

- **Point-to Analysis**
  - Whole-program analysis to find all possible targets

- **Cons**
  - Precise analysis can't scale
  - Suffers from soundness or precision issues
  - Itself requires a call-graph
Two approaches: Point-to analysis vs. Type analysis

- **Point-to Analysis**
  - Whole-program analysis to find all possible targets

- **Cons**
  - Precise analysis can't scale
  - Suffers from soundness or precision issues
  - Itself requires a call-graph

- **(First-Layer) Type Analysis**
  - Matching types of functions and function pointers (FLTA)

- **Cons**
  - Over-approximate
  - Worse precision in larger programs
Two approaches: Point-to analysis vs. Type analysis

● Point-to Analysis
  ○ Whole-program analysis to find all possible targets

● Cons
  ○ Precise analysis can't scale
  ○ Suffers from soundness or precision issues
  ○ Itself requires a call-graph

● (First-Layer) Type Analysis
  ○ Matching types of functions and function pointers (FLTA)

● Cons
  ○ Over-approximate

Practical and used by CFI techniques
Our intuition:

Function addresses are often stored to structs layer by layer.

Layered type matching is much stricter.
Our intuition:

Function addresses are often stored to structs layer by layer.

MLTA: Multi-Layer Type Analysis
// Assign address of foo to a nested field
1. a->b->c->fptr = &foo;
2. d->b->c->fptr = &bar;
   ... // Complicated data flow
3. a->b->c->fptr(10); // Indirect call to foo() not bar()
// Assign address of foo to a nested field
1. a->b->c->fptr = &foo;
2. d->b->c->fptr = &bar;
   ... // Complicated data flow
3. a->b->c->fptr(10); // Indirect call to foo() not bar()
// Assign address of foo to a nested field
1. a->b->c->fptr = &foo;
2. d->b->c->fptr = &bar;
   ... // Complicated data flow
3. a->b->c->fptr(10); // Indirect call to foo() not bar()
Illustrate MLTA

// Assign address of foo to a nested field
1. a->b->c->fptr = &foo;
2. d->b->c->fptr = &bar;
   ... // Complicated data flow
3. a->b->c->fptr(10); // Indirect call to foo() not bar()
Illustrate MLTA

// Assign address of foo to a nested field
1. a->b->c->fptr = &foo;
2. d->b->c->fptr = &bar;
   ... // Complicated data flow
3. a->b->c->fptr(10); // Indirect call to foo() not bar()

Layered type

<table>
<thead>
<tr>
<th>Layered type</th>
<th>&amp;foo</th>
<th>fptr()</th>
</tr>
</thead>
<tbody>
<tr>
<td>fptr_t</td>
<td>fptr</td>
<td>fptr</td>
</tr>
<tr>
<td>struct C</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>struct B</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>struct A</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

Complicated data flow
Illustrate MLTA

// Assign address of foo to a nested field
1. a->b->c->fptr = &foo;
2. d->b->c->fptr = &bar;
  ... // Complicated data flow
3. a->b->c->fptr(10); // Indirect call to foo() not bar()

Layered type

- &foo
- fptr
- fptr(10)

Only functions whose addresses are ever stored to the layered type can be valid targets.
Results comparison of approaches

// Assign address of foo to a nested field
1. a->b->c->fptr = &foo;
2. d->b->c->fptr = &bar;
   ... // Complicated data flow
3. a->b->c->fptr(10); // Indirect call to foo() not bar()

<table>
<thead>
<tr>
<th>Approach</th>
<th>MLTA</th>
<th>FLTA</th>
<th>2-Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matched targets</td>
<td>foo()</td>
<td>foo(), bar()</td>
<td>foo(), bar()</td>
</tr>
</tbody>
</table>
Advantages of the MLTA approach

- Most function addresses are stored to structs
  - 88% in the Linux kernel

- Being elastic
  - When a lower layer is unresolvable, fall back
  - Avoid false negatives

- MLTA should be always better than FLTA
- No expensive or error-prone analysis
“This is very intuitive; what are the challenges?”

“Fine-grained control-flow integrity for kernel software” (EuroSP’16) by Xinyang Ge, Nirupama Talele, Mathias Payer, Trent Jaeger.
Research questions and challenges

- To what extent can MLTA refine the targets?
- Can MLTA guarantee soundness?
  - No false negatives
- Can MLTA also support C++?
  - Virtual functions and tables
- Can MLTA scale to large and complex programs?
- How can MLTA benefit static analysis and bug finding?
Our technical contributions

- Multiple techniques to ensure effectiveness and soundness
  - With an elastic design and formal analysis
- Support C++
- Extensive evaluation (OS kernels and a browser)
- 35 new kernel security bugs
Realize MLTA: Overview of the TypeDive system

- **Phase I: Layered type analysis**
  - Three analysis techniques and three data structures
- **Phase II: Indirect-call targets resolving**
  - An iterative and elastic algorithm
Analyze type-function confinements

- **Purpose**
  - To identify which *types* have been assigned with which *functions*
  - We say *type A confines foo()* if &*foo* is stored to an A object

- **Inputs**
  - Address-taking and -storing operations
  - Global object initializers

- **Output**
  - The type-function confinement map
Analyze type-function confinements

- **Purpose**
  - To identify which **types** have been assigned with which **functions**
  - We say type A confines `foo()`, if `&foo` is stored to an A object

- **Inputs**
  - Address-taking and -storing operations
  - Global object initializers

- **Output**
  - The type-function confinement map

```
1. a->fptr = &foo;
   ...
2. fptr1 = &bar;
```
Analyze type-function confinements

- **Purpose**
  - To identify which types have been assigned with which functions
  - We say type A confines foo(), if &foo is stored to an A object

- **Inputs**
  - Address-taking and -storing operations
  - Global object initializers

- **Output**
  - The type-function confinement map

<table>
<thead>
<tr>
<th>Type</th>
<th>Function set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fptr_t</td>
<td>foo(), bar()</td>
</tr>
<tr>
<td>struct A&lt;fptr_t&gt;</td>
<td>foo()</td>
</tr>
</tbody>
</table>

1. a->fptr = &foo;
   ...
2. fptr1 = &bar;
Analyze type propagations

- **Purpose**
  - To capture propagation of addresses from one type to another

- **Inputs**
  - Type casts and non-address-taking object stores

- **Output**
  - The type-propagation map
Analyze type propagations

● Purpose
  ○ To capture propagation of addresses from one type to another

● Inputs
  ○ Type casts and non-address-taking object stores

● Output
  ○ The type-propagation map

1. a = (struct A*)b;
   ...
2. c->a = a;
Analyze type propagations

● Purpose
  ○ To capture propagation of addresses from one type to another

● Inputs
  ○ Type casts and non-address-taking object stores

● Output
  ○ The type-propagation map

1. a = (struct A*)b;
   ...
2. c->a = a;

<table>
<thead>
<tr>
<th>Destination type</th>
<th>Source type</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct A</td>
<td>struct B</td>
</tr>
<tr>
<td>struct C_A</td>
<td>struct A</td>
</tr>
</tbody>
</table>
Analyze type propagations

● Purpose
  ○ To capture propagation of addresses from one type to another

● Inputs
  ○ Type casts and non-address-taking object stores

● Output
  ○ The type-propagation map

1. a = (struct A*)b;
   ...
2. c->a = a;

<table>
<thead>
<tr>
<th>Destination type</th>
<th>Source type</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct A</td>
<td>struct B</td>
</tr>
<tr>
<td>struct C_A</td>
<td>struct A</td>
</tr>
</tbody>
</table>

Only for non-confinement stores
Identify escaped types

● **Purpose**
  ○ To identify types that may hold *undecidable* functions
  ○ Discard such types to avoid false negatives

● **What conditions result in an escaped type?**

**Unsupported type:**
1. General pointer (e.g., char *) and integer types *or*
2. Types with arithmetically computed object pointers

**A type is escaping if:**
1. It is cast from an unsupported type *or*
2. It is cast to an unsupported type
Examples of escaping cases

// Case 1
void * ptr = ...;
...
c->a = (struct A*)ptr;

// Case 2
void *ptr = (void *)c->a;
Resolve indirect-call targets

Maintained data structures
- Type-function map
- Type-propa. map
- Escaped types

Targets resolving
Resolve indirect-call targets

**Targets resolving**
For each indirect call, do initialization

- Maintained data structures
  - Type-function map
  - Type-propa. map
  - Escaped types
Resolve indirect-call targets

Maintained data structures
- Type-function map
- Type-propa. map
- Escaped types

Targets resolving
For each indirect call, do initialization
Get current layered type
Resolve indirect-call targets

Maintained data structures:
- Type-function map
- Type-propa. map
- Escaped types

Targets resolving:
For each indirect call, do initialization
- Get current layered type
- Escaped type?
Resolve indirect-call targets

Maintained data structures
- Type-function map
- Type-propa. map
- Escaped types

Targets resolving
For each indirect call, do initialization
Get current layered type
Escaped type?
No
Get next layer?
Resolve indirect-call targets

Maintained data structures
- Type-function map
- Type-propa. map
- Escaped types

Targets resolving
For each indirect call, do initialization
- Get current layered type
- Escaped type?
  - No
  - Get next layer?
    - Yes
    - Get current layered type
Resolve indirect-call targets

Maintained data structures
- Type-function map
- Type-propa. map
- Escaped types

Targets resolving
For each indirect call, do initialization
- Get current layered type
- Escaped type?
  - Yes: Go prev layer
  - No: Get next layer?
    - Yes: Go prev layer
    - No: Escaped type?
Resolve indirect-call targets

Maintained data structures
- Type-function map
- Type-propa. map
- Escaped types

Targets resolving
For each indirect call, do initialization
- Get current layered type
- Escaped type?
  - Yes: Go prev layer
  - No: Get next layer?
    - Yes: Escaped type?
    - No: Targets resolving
Resolve indirect-call targets

Maintained data structures
- Type-function map
- Type-propa. map
- Escaped types

For each indirect call, do initialization
- Get current layered type
- Escaped type?
  - Yes: Leave
  - No: Get next layer?
    - Yes: Recursively resolve targets for the layered type
    - No: Go prev layer

Indirect-call targets
Resolve indirect-call targets

The recursive resolving algorithm queries type-function and type-propagation maps to collect all targets.
Support C++

- **Problem:** VTable pointers are always cast to unsupported-type pointers
  - Identified as escaped types
  - Cannot benefit from MLTA at all

- **Our solution:** Directly map virtual functions to class types by skipping VTable pointers
  - Also support multiple inheritances
Implementation

- Based on LLVM
- Supported types: `struct`, `vector`, and `function type`
- Field-sensitive, but flow-insensitive and context-insensitive
- Hashing type information to reduce memory overhead
Formal analysis of effectiveness and soundness

<table>
<thead>
<tr>
<th>confinement</th>
<th>propagation</th>
<th>resolving</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLTA</td>
<td>FLTA</td>
<td>MLTA</td>
</tr>
<tr>
<td>( a = &amp;f )</td>
<td>( y = \text{cast}&lt;t(y)&gt;x )</td>
<td>( (*p)() )</td>
</tr>
<tr>
<td>( M[t(a)] \cup = {f} )</td>
<td>( M[t(y)] \cup = M[t(x)] )</td>
<td>( M[t(p)] )</td>
</tr>
<tr>
<td>MLTA</td>
<td>MLTA</td>
<td>MLTA</td>
</tr>
<tr>
<td>( a = &amp;f )</td>
<td>( y = x )</td>
<td>( (*p)() )</td>
</tr>
<tr>
<td>( M[mlt(a)] \cup = {f} )</td>
<td>( \forall \alpha \in \text{comp}(mlt(y)), \forall \beta \in \text{comp}(mlt(x)), M[mlt(y)] \cup = M[\beta], M[\alpha] \cup = M[\beta] )</td>
<td>( \forall y \in \text{comp}(mlt(p)) \cup M[y] )</td>
</tr>
</tbody>
</table>

We prove:

- MLTA has fewer FPs than FLTA (**effectiveness**)
- FLTA may have FNs, but MLTA does not introduce extra FNs (**soundness**)

Details in the paper
Evaluate MLTA

• Evaluation goals
  ○ Scalability, effectiveness, soundness, and use cases

• Experimental setup
  ○ The Linux kernel, the FreeBSD kernel, and the Firefox browser
  ○ 64GB RAM and Intel CPU (3.20 GHz, 8 cores)

<table>
<thead>
<tr>
<th>System</th>
<th>Modules</th>
<th>SLoC</th>
<th>Loading Time</th>
<th>Analysis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>17,558</td>
<td>10,330K</td>
<td>2m 6s</td>
<td>1m 40s</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>1,481</td>
<td>1,232K</td>
<td>6s</td>
<td>6s</td>
</tr>
<tr>
<td>Firefox</td>
<td>1,541</td>
<td>982K</td>
<td>27s</td>
<td>1m 25s</td>
</tr>
</tbody>
</table>
Reduction of indirect-call targets: Average number

- MLTA-eligible indirect calls: 81%, 64%, 63%
- MLTA achieves 94%, 86%, 98% further reduction over FLTA
- The second layer achieves the most reduction
- More layers keep reducing the number
  - 5 layers suffice
Reduction of indirect-call targets: Distribution (Linux)

- <8 targets: MLTA 89%, FLTA 58%
- Largest number: MLTA 1,914 targets, FLTA 7,983 targets
False-negative analysis

Trace execution to collect “ground-truth” targets

- Instrument Firefox with PTWRITE via LLVM pass
  - Dump source & destination for each indirect call
  - \(50k\) pairs of \(<\text{indirect call, callee}>\)
- Run Linux in QEMU and hook indirect calls
  - Hook \(__x86\)_indirect_thunk_rax
  - \(3,566\) pairs of \(<\text{indirect call, callee}>\)
- Several FNs caused by FLTA or lacking source
False-negative analysis

Trace execution to collect “ground-truth” targets

- Instrument Firefox with PTWRITE via LLVM pass
  - Dump source & destination for each indirect call
  - 50k pairs of $<\text{indirect call, callee}>$
- Run Linux in QEMU and hook indirect calls
  - Hook $\text{__x86_indirect_thunk_rax}$
  - 3,566 pairs of $<\text{indirect call, callee}>$
- Several FNs caused by FLTA or lacking source

The MLTA approach does not introduce extra false negatives than FLTA
### Benefit static-analysis and bug-finding

10 uninitialization bugs (see the left table)
- FLTA #func → MLTA #func
- MLTA helps save efforts

25 missing-check bugs (see the paper)

<table>
<thead>
<tr>
<th>[Subsys] File</th>
<th>Function</th>
<th>Variable</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>drivers/gpu/drm/gma500/oaktrail_crtc.c:511</td>
<td>[13-&gt;3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[drm] cdv_intel_display.c</td>
<td>cdv_intel_find_dp_pll</td>
<td>clock</td>
<td>4B</td>
</tr>
<tr>
<td>[drm] oaktrail_crtc.c</td>
<td>mrst_sdvo_find_best_pll</td>
<td>clock</td>
<td>16B</td>
</tr>
<tr>
<td>[drm] oaktrail_crtc.c</td>
<td>mrst_lvds_find_best_pll</td>
<td>clock</td>
<td>16B</td>
</tr>
<tr>
<td>drivers/media/v412-core/v412-ioctl.c:1509</td>
<td>[438-&gt;5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[media] rcar_drif.c</td>
<td>rcar_drif_g_fmt_sdr_cap</td>
<td>f</td>
<td>24B</td>
</tr>
<tr>
<td>drivers/staging/rtl18188eu/core/rtw_security.c:229</td>
<td>[18-&gt;6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[crypto] lib80211_crypt_wep.c</td>
<td>lib80211_wep_set_key</td>
<td>wep</td>
<td>25B</td>
</tr>
<tr>
<td>[staging] rttlib_crypt_wep.c</td>
<td>prism2_wep_set_key</td>
<td>wep</td>
<td>25B</td>
</tr>
<tr>
<td>drivers/staging/media/davinci_vpfe/dm365_ipipe.c:1277</td>
<td>[36-&gt;18]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[staging] dm365_ipipe.c</td>
<td>ipipe_set_wb_params</td>
<td>wbal</td>
<td>8B</td>
</tr>
<tr>
<td>[staging] dm365_ipipe.c</td>
<td>ipipe_set_rgb2rgb_params</td>
<td>rgb2rgb_</td>
<td>12B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>defaults</td>
<td></td>
</tr>
<tr>
<td>[staging] dm365_ipipe.c</td>
<td>ipipe_set_rgb2yuv_params</td>
<td>rgb2yuv_</td>
<td>4B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>defaults</td>
<td></td>
</tr>
<tr>
<td>crypto/af_alg.c:302</td>
<td>[13-&gt;3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[crypto] algif_hash.c</td>
<td>hash_accept_parent_nokey</td>
<td>ctx</td>
<td>680B</td>
</tr>
</tbody>
</table>
Conclusions

● MLTA can dramatically refine indirect-call targets
  ○ Multiple new techniques and formal analysis
  ○ 86%-98% further reduction over FLTA
  ○ Scale to large systems and support C/C++
  ○ No extra false negatives

● A building block for static analysis and CFI

● Precise indirect-call targets can serve as peers for detecting deep bugs
  ○ Identify deviating operations