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

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

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

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

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.

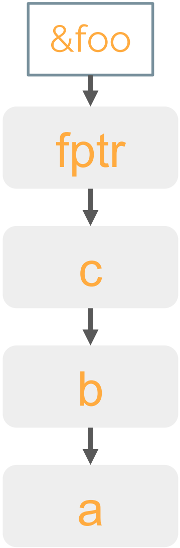
MLTA: Multi-Layer Type Analysis

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()
```

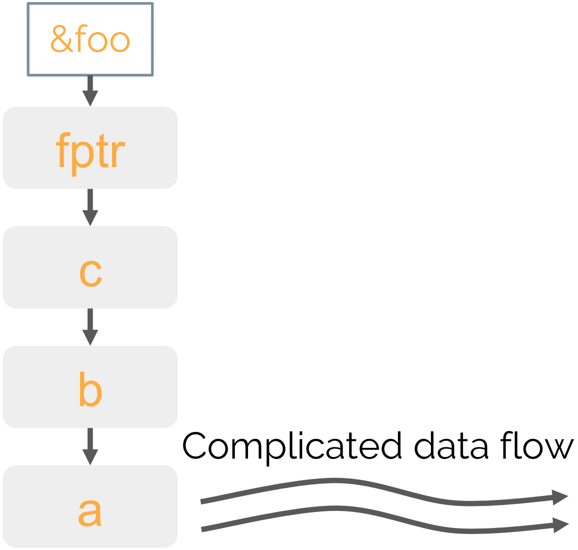
Illustrate MLTA

```
   // 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()
```



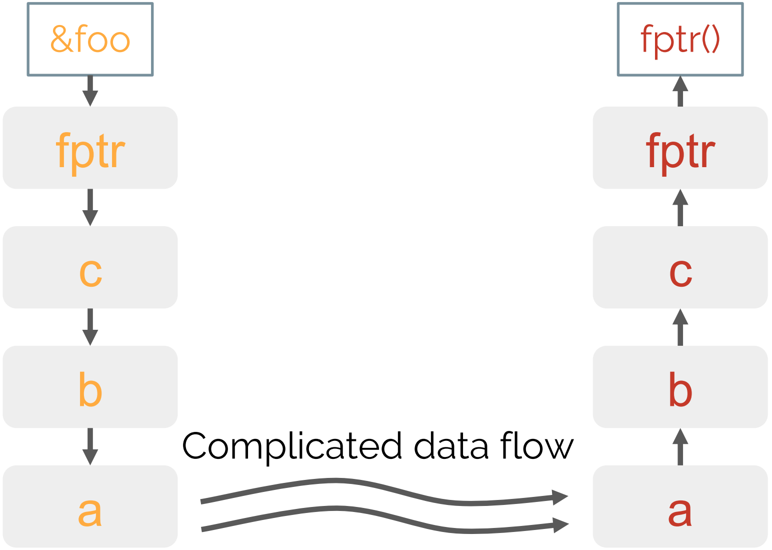
Illustrate MLTA

```
    // 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()
```



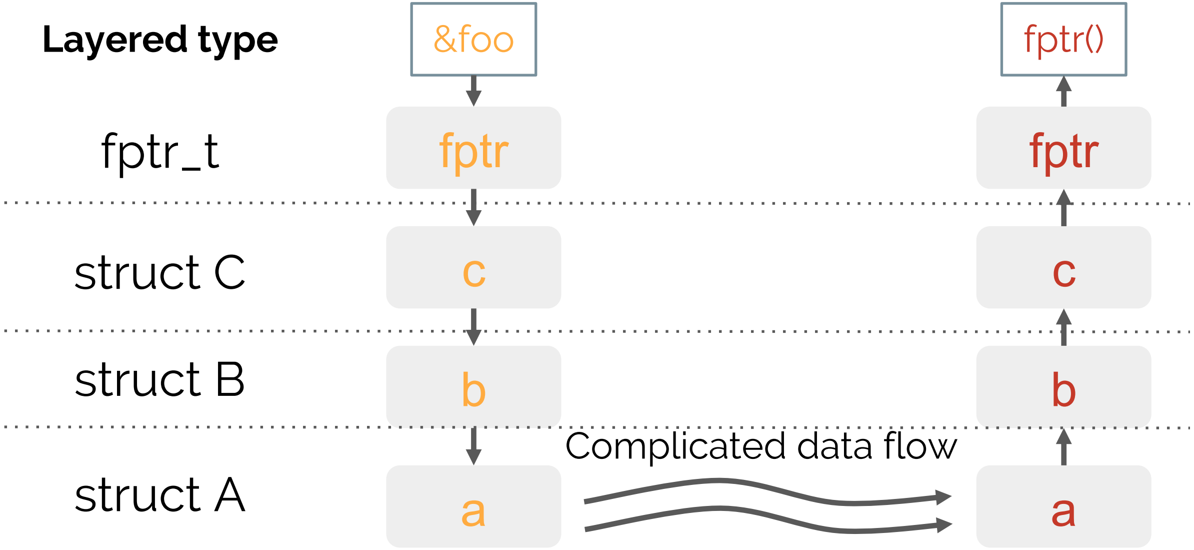
Illustrate MLTA

```
   // 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()
```



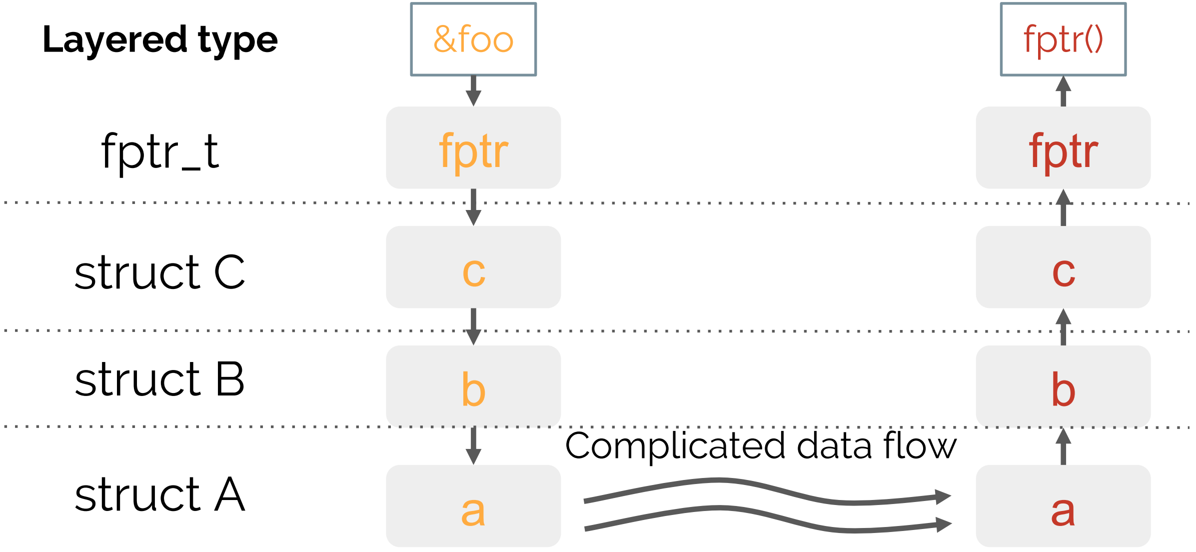
Illustrate MLTA

```
   // 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()
```



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()
```

Approach	**MLTA**	**FLTA**	**2-Layer**
Matched targets	foo()	foo(), bar()	foo(), bar()

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.

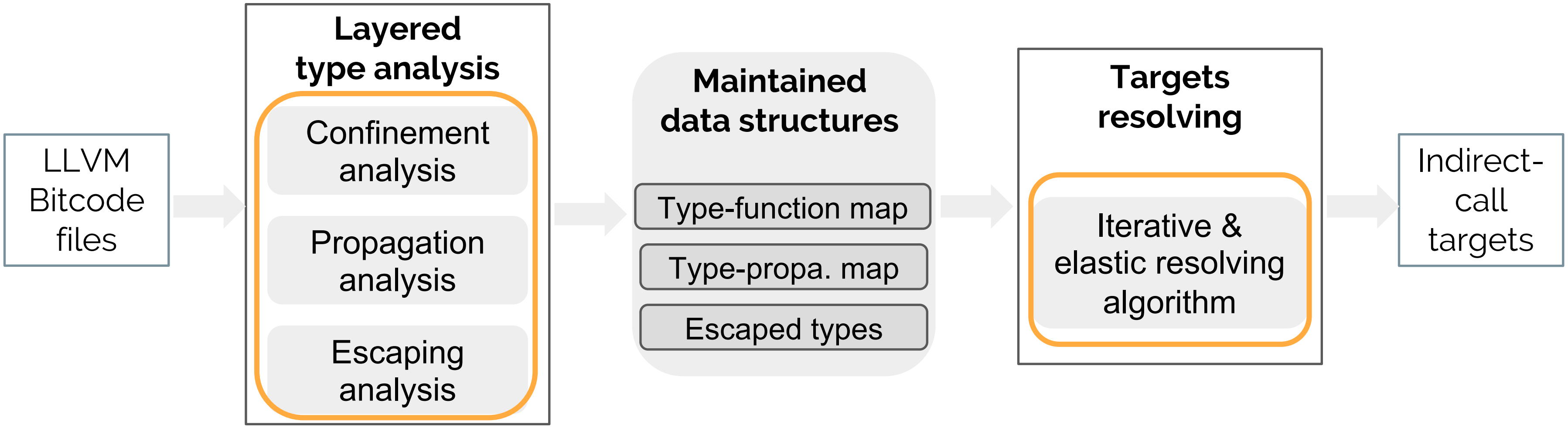
Research questions and challenges

- 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

```
1. a->fptr = &foo;
   ...
2. fptr1 = &bar;
```

Type	Function set
fptr_t	foo(), bar()
struct A_{fptr_t}	foo()

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;
```

Destination type	**Source type**
struct A	struct B
struct C_A	struct 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;
```

Only for non-confinement stores

Destination type	Source type
struct A	struct B
struct C_A	struct A

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***

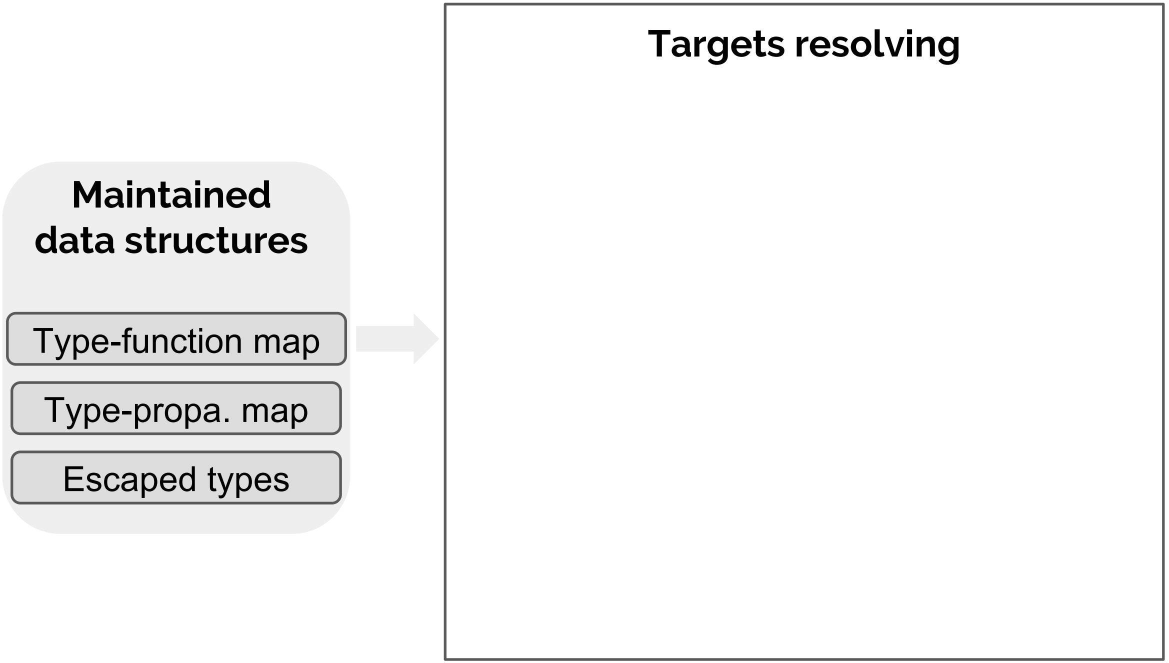
(2) It is cast to an unsupported type

Examples of escaping cases

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

```
// Case 2
void *ptr = (void *)c->a;
```

Resolve indirect-call targets



Resolve indirect-call targets

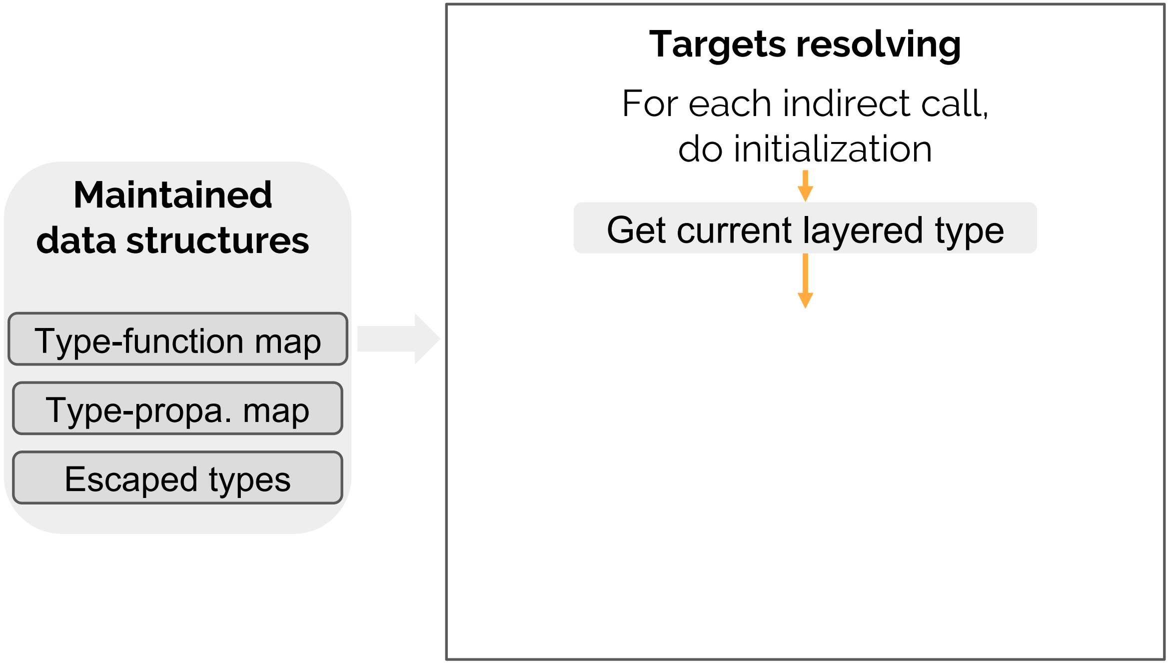
Maintained data structures

- Type-function map
- Type-propa. map
- Escaped types

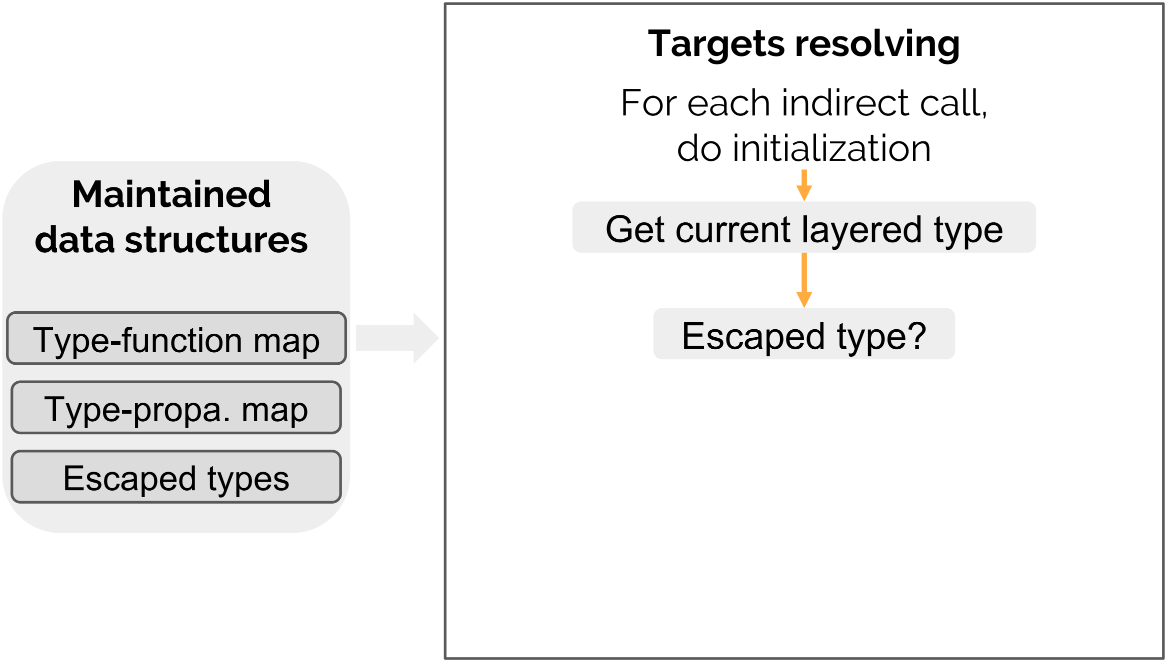
Targets resolving

For each indirect call,
do initialization

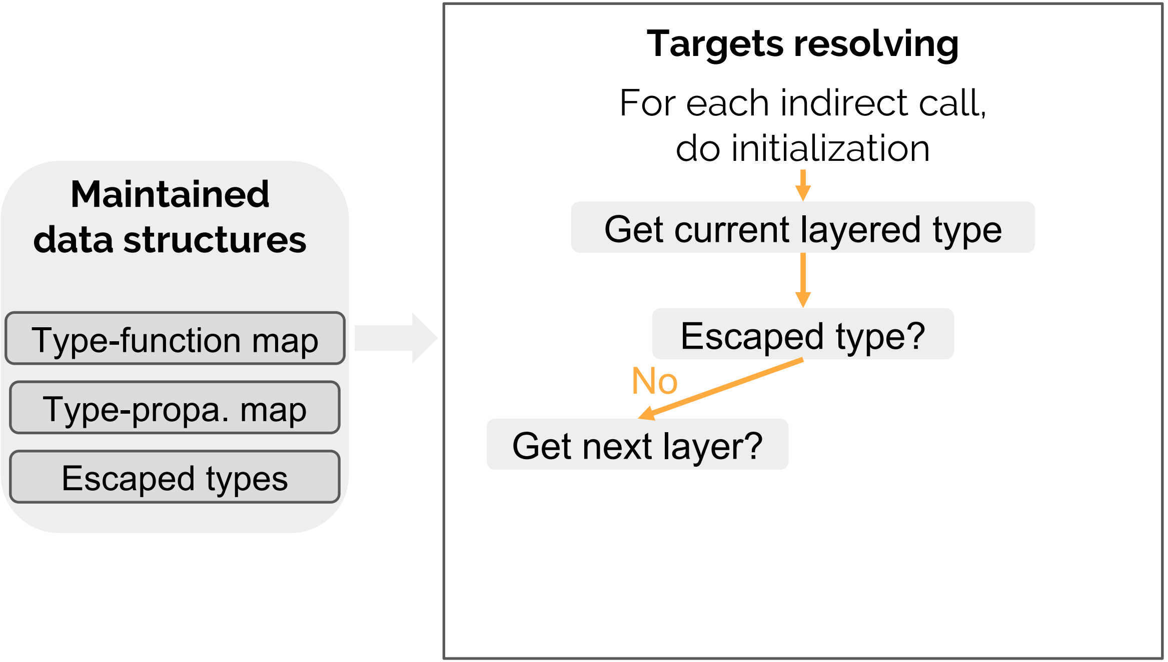
Resolve indirect-call targets



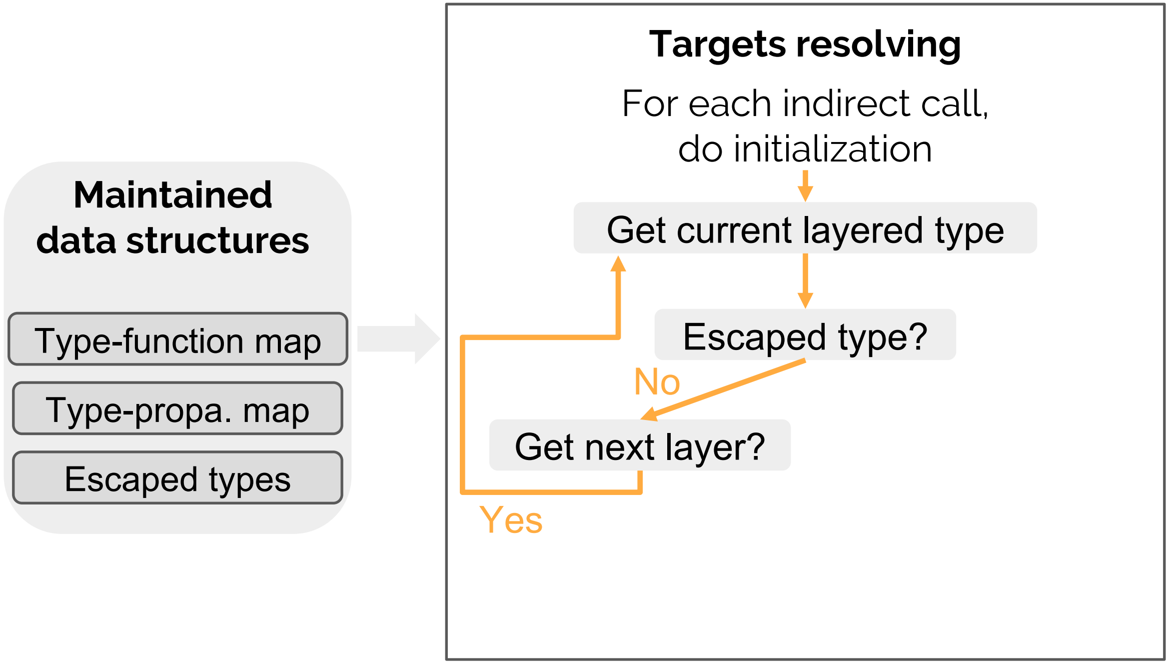
Resolve indirect-call targets



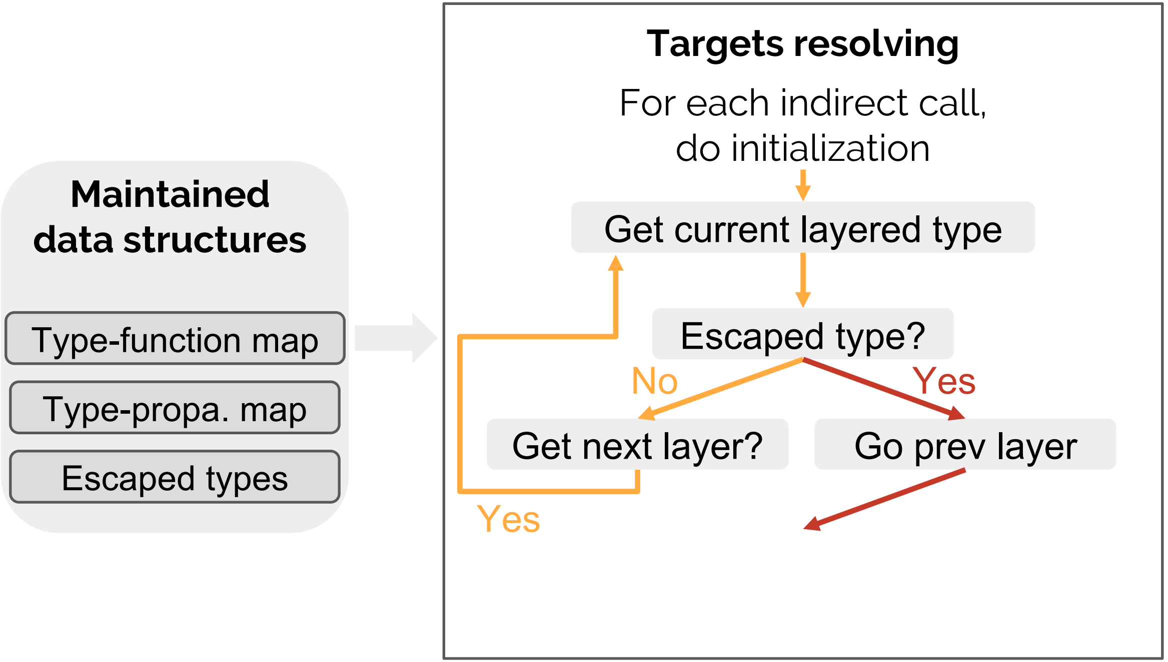
Resolve indirect-call targets



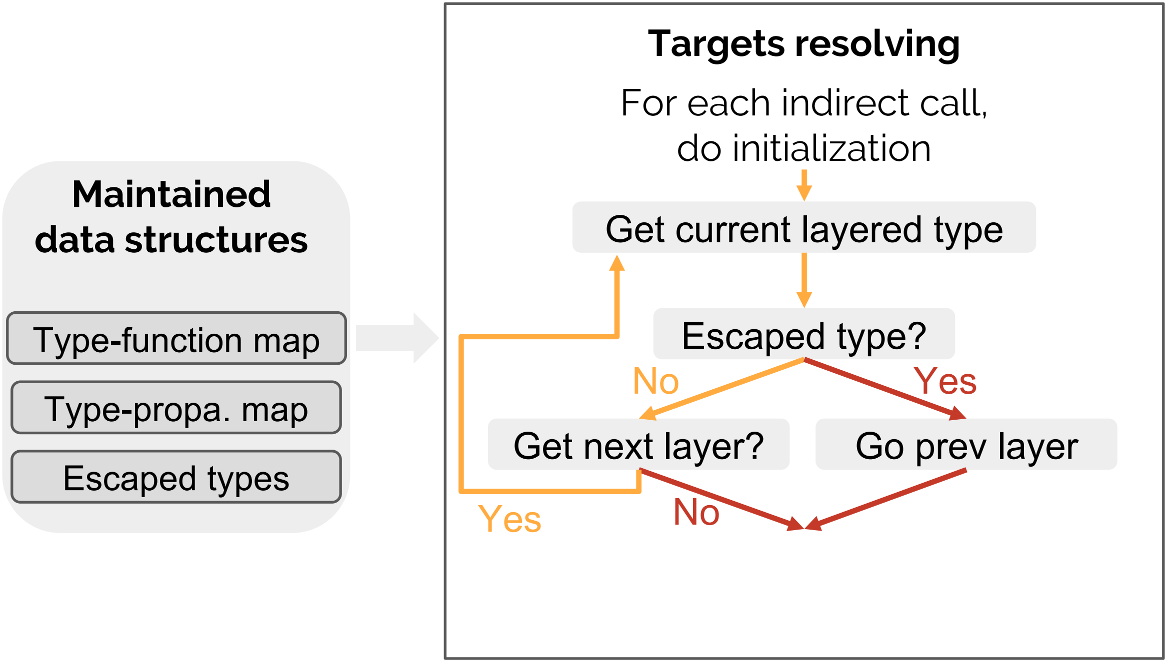
Resolve indirect-call targets



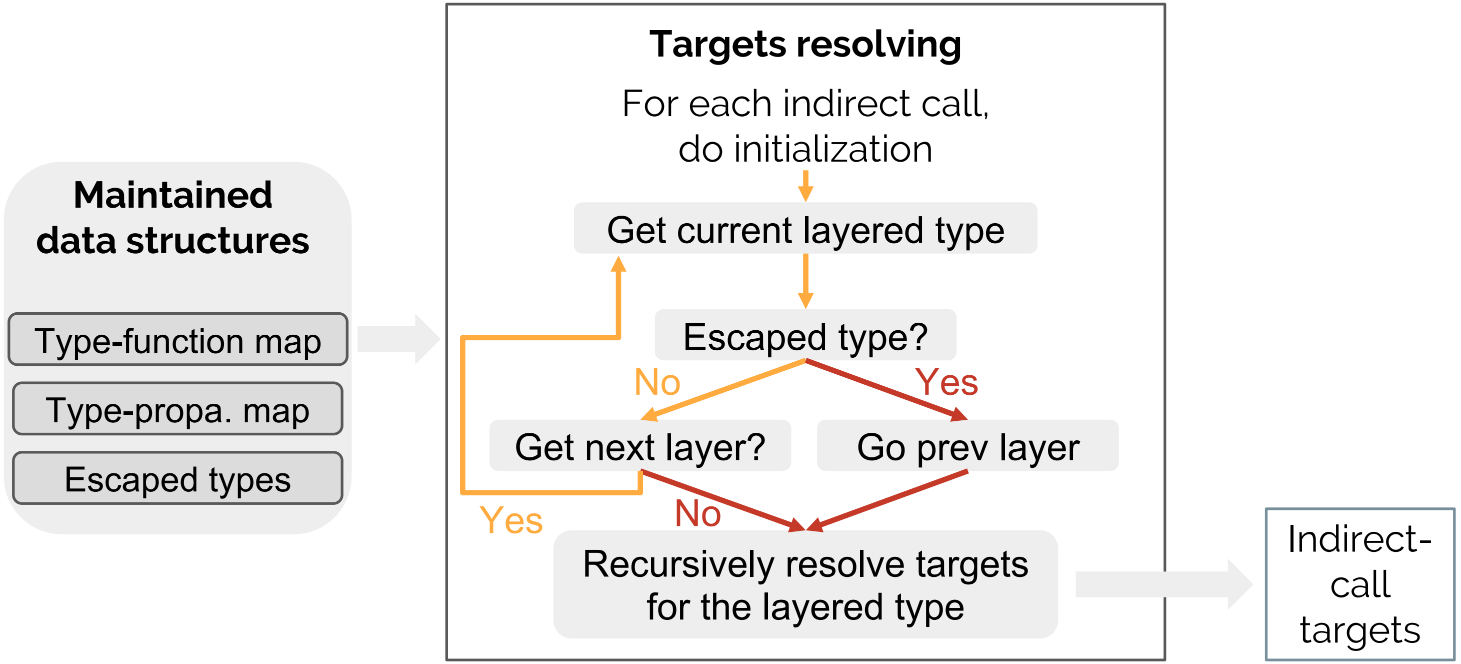
Resolve indirect-call targets



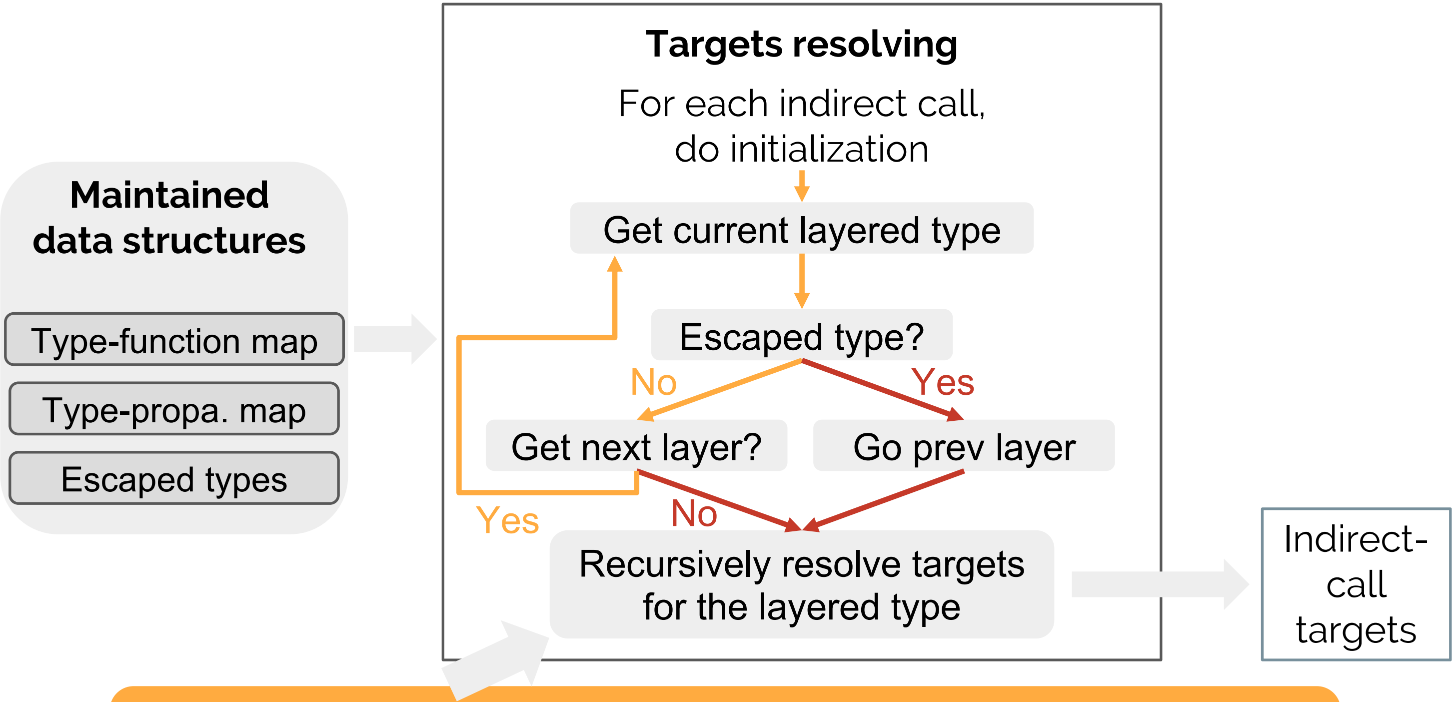
Resolve indirect-call targets



Resolve 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

	confinement	propagation	resolving
FLTA	$\dfrac{a = \&f}{M[t(a)] \cup= \{f\}}$	$\dfrac{y = cast{<}t(y){>}\ x}{M[t(y)] \cup= M[t(x)]}$	$\dfrac{(*p)()}{M[t(p)]}$
MLTA	$\dfrac{a = \&f}{M[mlt(a)] \cup= \{f\}}$	$\dfrac{y = x}{\begin{array}{c}\forall\alpha \in comp(mlt(y)),\\ \forall\beta \in comp(mlt(x)),\\ M[mlt(y)] \cup= M[\beta]\\ M[\alpha] \cup= M[\beta]\end{array}}$	$\dfrac{(*p)()}{\begin{array}{c}\forall\gamma \in comp(mlt(p))\\ \bigcup \mathrm{M}[\gamma]\end{array}}$

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

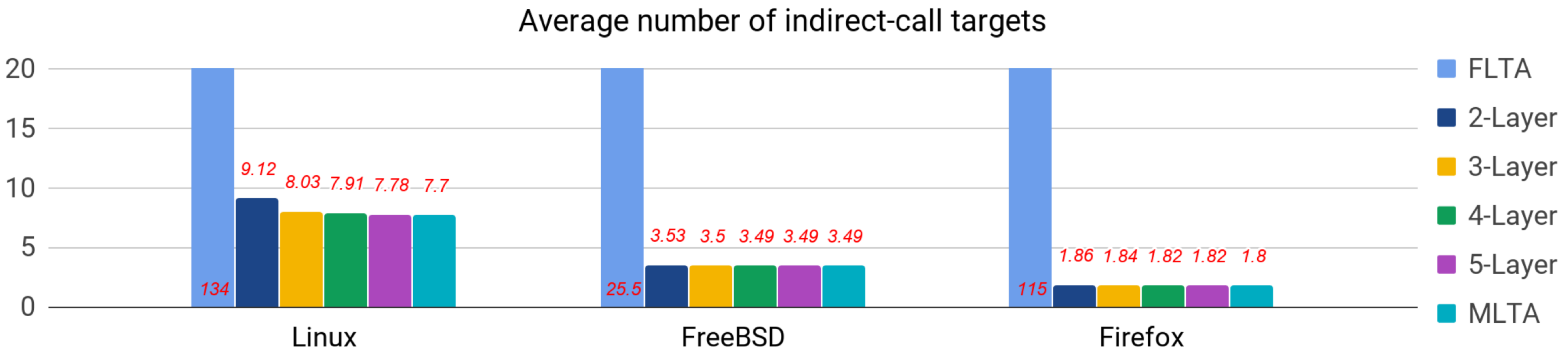
Evaluate MLTA

- 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)

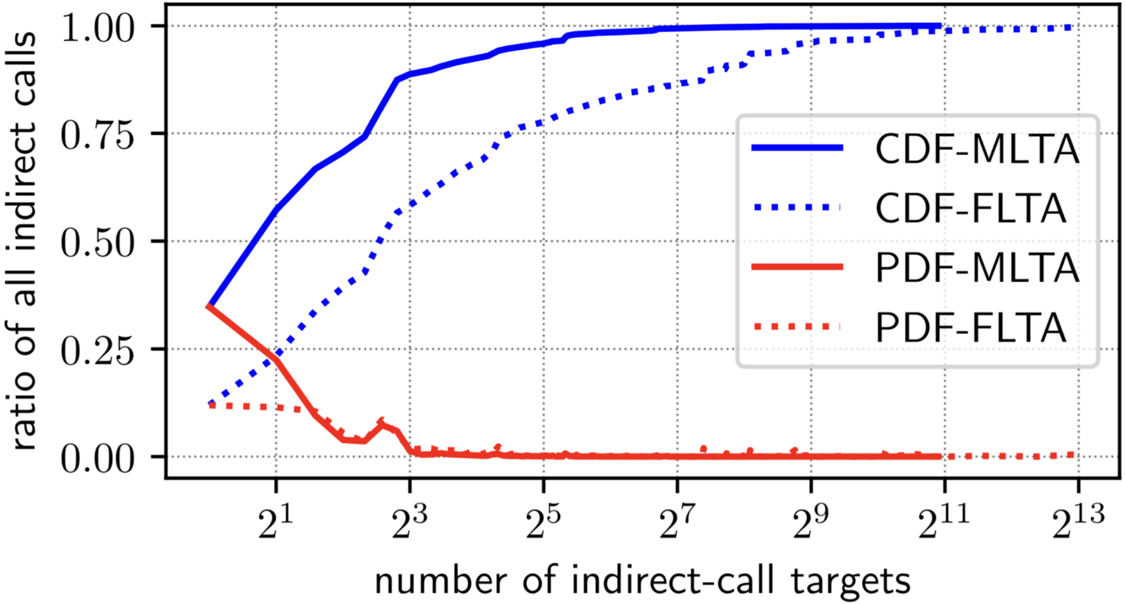
System	**Modules**	**SLoC**	**Loading Time**	**Analysis Time**
Linux	17,558	10,330K	2m 6s	1m 40s
FreeBSD	1,481	1,232K	6s	6s
Firefox	1,541	982K	27s	1m 25s

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 <*indirect call, callee*>
- Run Linux in QEMU and hook indirect calls
  - Hook __x86_indirect_thunk_rax
  - **3,566** pairs of <*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 <*indirect call, callee*>
- Run Linux in QEMU and hook indirect calls
  - Hook __x86_indirect_thunk_rax
  - **3,566** pairs of <*indirect call, callee*>
- Sev

The MLTA approach does not introduce extra false negatives than FLTA

Benefit static-analysis and bug-finding

[Subsys] File	Function	Variable	Impact	
drivers/gpu/drm/gma500/oaktrail_crtc.c:511 [13->3]				
[drm] cdv_intel_display.c	cdv_intel_find_dp_pll	clock	4B	UI
[drm] oaktrail_crtc.c	mrst_sdvo_find_best_pll	clock	16B	LK
[drm] oaktrail_crtc.c	mrst_lvds_find_best_pll	clock	16B	LK
drivers/media/v4l2-core/v4l2-ioctl.c:1509 [438->5]				
[media] rcar_drif.c	rcar_drif_g_fmt_sdr_cap	f	24B	UI
drivers/staging/rtl8188eu/core/rtw_security.c:229 [18->6]				
[crypto] lib80211_crypt_wep.c	lib80211_wep_set_key	wep	25B	UI
[staging] rtllib_crypt_wep.c	prism2_wep_set_key	wep	25B	UI
drivers/staging/media/davinci_vpfe/dm365_ipipe.c:1277 [36->18]				
[staging] dm365_ipipe.c	ipipe_set_wb_params	wbal	8B	UI
[staging] dm365_ipipe.c	ipipe_set_rgb2rgb_params	rgb2rgb_ defaults	12B	UI
[staging] dm365_ipipe.c	ipipe_set_rgb2yuv_params	rgb2yuv_ defaults	4B	UI
crypto/af_alg.c:302 [13->3]				
[crypto] algif_hash.c	hash_accept_parent_nokey	ctx	680B	UI

10 uninitialization bugs

(see the left table)

- FLTA #func → MLTA #func
- MLTA helps save efforts

25 missing-check bugs

(see the paper)

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