Improving Indirect-Call Analysis in LLVM with Type and Data-Flow Co-Analysis

Mehdi Zaidi, Peter Cao, Amr Hussein, Mohammadali Kodeih

November 10, 2025

MOTIVATION

Indirect-Call Primer

- Label call: Regular functional call, know where to jump to during compile-time
  - func(...);
- Address call: Function call taken from address during compile-time
  - Indirect calls make it difficult to construct CFGs precisely
  - Can be sources of bugs and security issues
  - void(*func_ptr)(int, int), std::function<void(int, int)>

Dynamic vs Static Analysis

Dynamic Analysis

- Build CFG at runtime
- Heavily depends on code coverage
- Limited soundness

Static Analysis

- Build CFG statically
- Large programs like OS Kernels make precise pointer analysis not feasible
- Practical to use type analysis to match signatures
  - Many false positives if many functions share signature

Prior Work: MLTA

- Multi-Layer Type Matching
- Takes advantage of the fact that many indirect calls are invoked from structs
- Outer structs used to differentiate signatures
  - Still ineffective for icalls addresses outside of structs
  - Room left for data flow analysis (TFA)

```c
typedef void (*f_ptr)(int a, int b);
struct S {f_ptr field1; f_ptr field2};

void address_taken_func1(int a, int b){...}
void address_taken_func2(int a, int b){...}
void address_taken_func3(int a, int b){...}
void address_taken_func4(int a, int b){...}

struct S s1 = {.field1 = address_taken_func1,
               .field2 = address_taken_func2};
struct S s2 = {.field1 = address_taken_func3,
               .field2 = address_taken_func4};

void main() {
    ...
    s1.field1(100, 200); // address_taken_func1 is called here
    ...
}
```

Figure 1: Example of type analysis for identifying icall targets.

Type Analysis Problems in LLVM

- **Missing function pointer fields**
  - Fields are empty pointers (known bug)
  - Need to populate these missing fields
- **Missing struct names**
  - Determining targets unclear if not initialized with declaration (see Figure 1)

```
/* source code */
struct A {
    int i;
    int (*f)(int, struct A*);
    int (*g)(char, struct A*);
};

/*Expected LLVM IR of struct A */
%struct.A = type {i32, i32 (i32, %struct.A*), i32 (i8, %struct.A*)}

/*Actual LLVM IR of struct A */
%struct.A = type {i32, {}*, i32 (i8, %struct.A*)}
```

Figure 2: Example of omitting function pointer fields.

Type Analysis Problems in LLVM

- **Type unfolding**
  - Types could be split into finer grained units
  - Common with union fields
- **Optimizations omit information**
  - GEP (get element ptr) instructions show the field within a struct
  - Compiling with -OX omits the information about the struct, disabling nested struct retrieval

```
/* source code */
struct dvb_usb_adapter_properties adapter[2];

/*Expected LLVM type of variable adapter */
[2 x %struct.dvb_usb_adapter_properties]

/*Actual LLVM type of variable adapter */
<{{i8, i8, i32 (%struct.dvb_usb_adapter*, i32)*,
i32 (%struct.dvb_usb_adapter*, i32, i16, i32)*, {i8, i8, i8,
{%struct.anon.163, [8 x i8]}}}, %struct.dvb_usb_adapter_properties}>
```

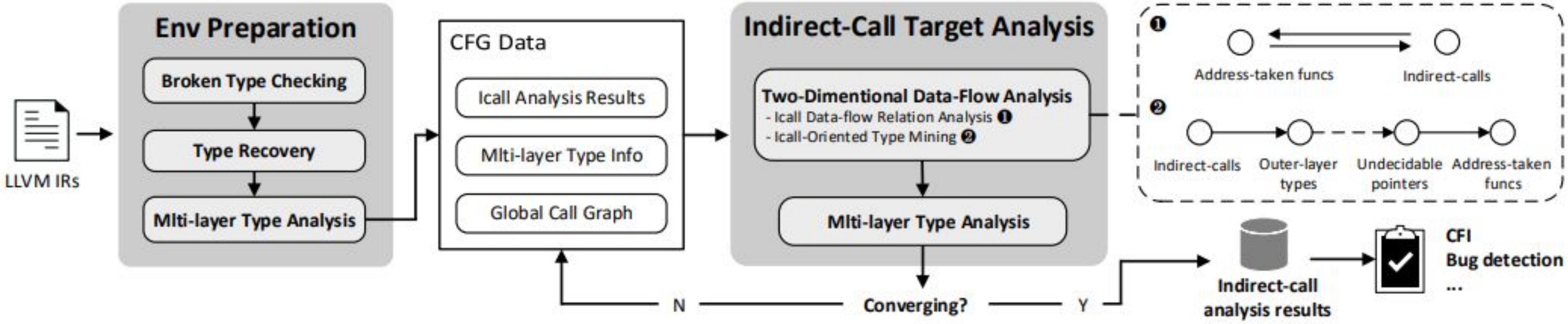
Figure 3: Example of type unfolding.

```
Before optimization:
%ptr = getelementptr inbounds %struct.S* %0, i64 0, i32 5
After optimization:
%ptr = getelementptr inbounds i8, i8* %0, i64 28
```

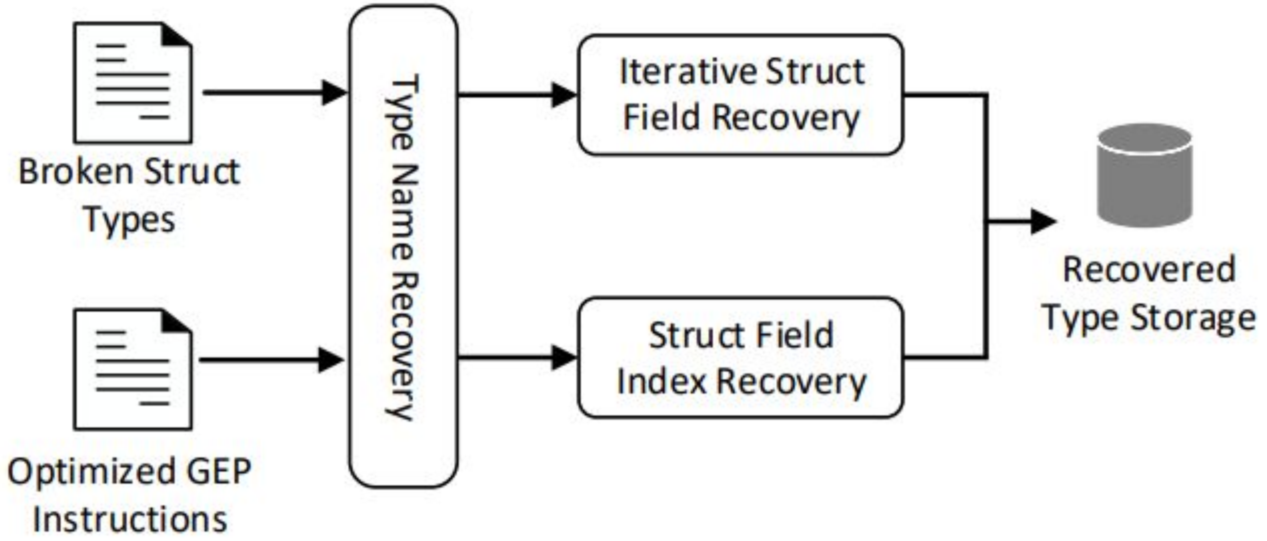
METHODOLOGY

Outline

- Type Recovery
- Two-Dimensional Data-Flow Analysis

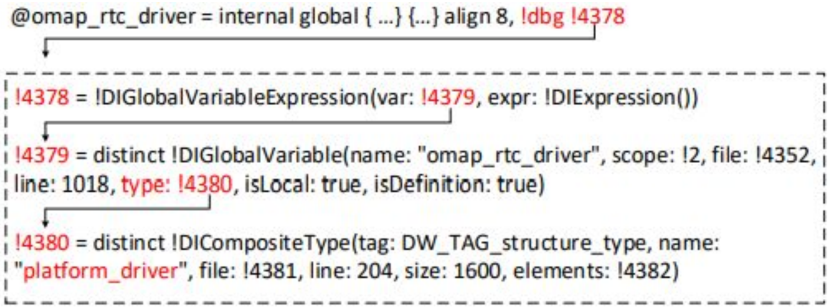


Type Recovery



Type Name Recovery

- Initial process before other two recovery processes
- For structs:
  - Unroll metadata layer-by-layer to get struct name
- For GEP instructions
  - Get pointer operand
  - Analyze use-chain to get llvm.dbg.value info
  - Get metadata and then unroll



Struct Field + Index Recovery

Field Recovery

- Used for recursive struct types
- Recursively use name recovery to recover inner struct names

Example:

global A {struct {type:!1}}

!1 = distinct !DICompositeType {name: B}

- 3 additional strategies for soundness
  - Unfold composite types -> primitive types
  - Int types -> multiple i8 types
  - void can cast to any pointer type

Field Index Recovery

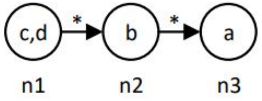
- For GEP, need to recover the field index (ie. for a.b.c -> need to recover the index of b and c)
  - Some analyses combine these into 1 GEP
- Recursively find offsets:
  - **Ex**: GEP a.b.c with offset 80
  - Offsets for a -> {0, 60, 120}
  - a.b will have offset 60 -> 80-60=20 remaining
  - Offsets for b -> {0, 5, 20, 40}
  - b.c has offset 20 -> a.b.c

Alias Graph

- Track variable alias relationships
- Features for soundness:
- **Inter-procedural**: Cross-function analysis
- **Flow-insensitive**: Alias can happen regardless of control-flow dependencies, do both forwards and backwards analysis
- **Field-insensitive**: MLTA already supports field-sensitivity
- **May-alias**: Conservative analysis instead if must-analysis

```
int a = 10;
int *b = &a;
int **c = &b;
int **d = &b;
```

Source code



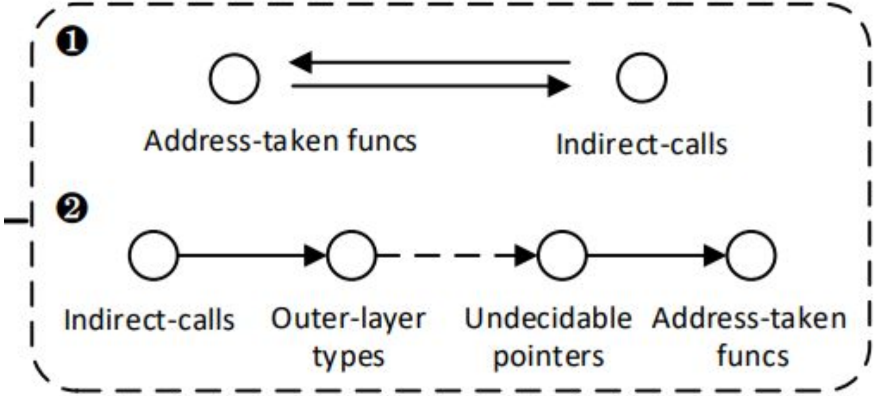
Alias graph

AliasSet1: {c,d}
AliasSet2: {b, *c, *d}
AliasSet3: {a, *b, **c, **d}

Alias sets

Two Dimensional Data-Flow Analysis

- Existing MLTA already minimizes false-positives
- 2-dimensions
  - Bidirectional Dataflow Relation Analysis
  - Lightweight type-information mining



Dataflow Relation Analysis

- **Bidirectional analysis**
  - *Backward*: Starts from an icall and identifies possible targets (intersect with MLTA results)
  - *Forward:* Starts from a target and find possible icalls
- Fallback to type analysis when false negatives are possible:
  - Alias values reaches some threshold
  - Values aliased with assembly code
  - Values aliased with arithmetic operations (ie. (*ptr1 + *ptr2)())

Type Mining

Type Mining for Icalls

- Run backwards analysis from destinations to their calls
- Determines whether a call is derived from a struct field, indicating aliasing
- Ex: Line 4, 16, and 14

```
1  /* sound/core/pcm_lib.c */
2  static int interleaved_copy(..., pcm_transfer_f transfer) {
3      ...
4      return transfer(...); //No outer-layer type in MLTA
5  }
6
7  snd_pcm_sframes_t __snd_pcm_lib_xfer(...) {
8      ...
9      pcm_copy_f writer;
10     pcm_transfer_f transfer;
11     ...
12     writer = interleaved_copy; //An icall of interleaved_copy
13     ...
14     transfer = substream->ops->copy_kernel;
15     ...
16     err = writer(..., transfer);
17 }
```

Figure 7: Example of hidden layered type.

Type Mining for Undecidible Targets

- Used when a struct field has an undecidable target (ie type-escaping such as from function args)
- MLTA uses a global map to track these
- Backwards data-flow to identify origin of the input pointer
- Ex: Line 3->16 from MLTA, Line 3->22 from TFA

```
1  /* drivers/gpu/drm/drm_aperture.c */
2  static int devm_aperture_acquire(struct drm_device *dev, ...
3              void (*detach)(struct drm_device *)) {
4      ...
5      struct drm_aperture *ap;
6      ...
7      ap->detach = detach; //An undetermined pointer is stored
8  }
9
10 static void drm_aperture_detach_drivers(resource_size_t base,
11             resource_size_t size) {
12     ...
13     struct drm_aperture *ap = container_of(...);
14     struct drm_device *dev = ap->dev;
15     ...
16     ap->detach(dev); //Indirect call site
17     ..
18 }
19
20 int devm_aperture_acquire_from_firmware(...) {
21     ...
22     return devm_aperture_acquire(...,
23         drm_aperture_detach_firmware);
24 }
```

BACKUP: Implementation

- LLVM 15, 12k lines of C++
- Bitcode generation: WriteBitCodeToFile to dump bitcode files
- Type Comparisson: LLVM uses union and anon types, TFA regards them as invalid -> 1 layer type matching fallback, only affects 0.4% of icall targets in Linux
- MLTA: TypeDive with 2 layer type matching
- EXPORT_SYMBOL results in macro-optimization and intermodule optimization (need to manually check source code to prevent IR looking like assembly code for premature termination)
- For virtual function calls: analyze constructurs to catalog virtual function tables, when we have a virtual function call, identify issuing class, and then get callee from VTable -> save resources

RESULTS/TESTING

Testing Goals

4 Questions asked -

- How does TFA compare to existing methods?
- Does TFA maintain the previous correctness?
- How do broken types affect TFA? How effective is recovery?
- Does it benefit real world applications?

Testing Setup

Test Environment:

- Ubuntu 20.04 LTS
- Uses Intel Xeon Silver 4316 (80c, 2.3ghz) with 126gb of ram

Benchmark Suite:

- **Linux Kernel** - Large-scale system software
- **FreeBSD Kernel** - Alternative OS implementation
- **OpenSSL** - Cryptographic library
- **OpenCV** - Computer vision library
- **MongoDB** - Database program

Testing Results

System	Language	Bitcode Files	Total Icalls	Avg. (Sig)	Avg. (MLTA)	Avg. (MLTA+VH)	Avg. (TFA)	Analysis Rounds	Analysis Time
OpenSSL	C	1,309	2,200	32.3	27.5	27.5	20.9 (24%↓)	2	34s
Linux-loc	C	2,978	9,527	52.5	18.6	18.6	8.3 (55%↓)	2	4m 8s
FreeBSD	C	3,826	20,901	38.1	20.2	20.2	11.6 (43%↓)	3	19m 4s
MongoDB	C++	4,406	23,885	34.8	30.0	11.7	6.6 (44%↓)	2	1h 57m
OpenCV	C++	1,583	33,602	44.5	44.5	32.6	14.2 (56%↓)	2	42m 2s
Linux-all	C	21,438	73,163	161.7	44.9	44.9	18.6 (59%↓)	3	1h 59m

- TFA resulted in significantly less (24 - 59%) potential jump locations per indirect function call when compared to the state of the art approach

Testing Results

Systems	Init	Round1	Round2	Round3
Linux-all	3,288,024	1,465,868	1,360,894	1,358,831
OpenSSL	60,417	46,224	46,038	-
MongoDB	279,272	158,971	158,177	-

Systems	BDA	TM-I	TM-UT
Linux-all	1,157,344	362,363	409,486
OpenSSL	7,204	1,501	5,674
MongoDB	94,968	8,633	17,494

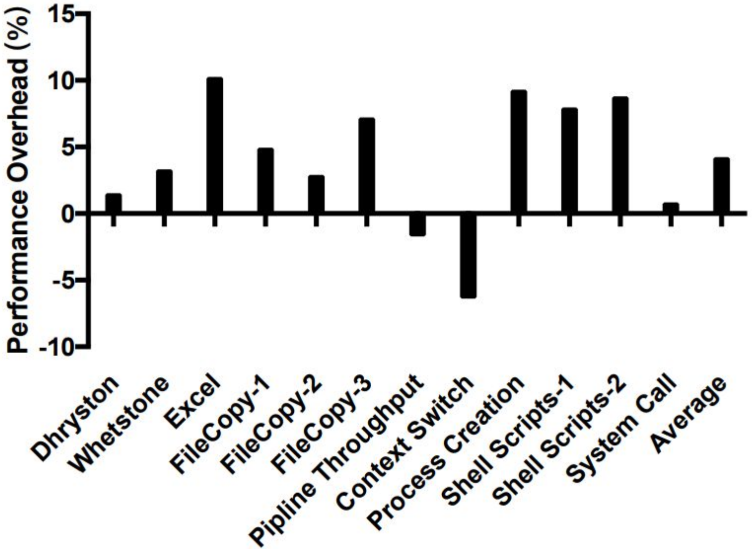
- Minimal reduction after 1 round.
- BDA reduces most of the icalls compared to TM-I and TM-UT
- Even with broken type information (TM-UT), TFA can still eliminate significant icall targets

False Negative Assessment

- Tested OpenSSL && Linux Kernel
- Linux had missed two potential jump locations
  - Suspect due to compiler optimizations
- OpenSSL had 58 false negatives
  - Type analysis issue (ex : void * versus int *)
  - Solved by allowing void * to be any type during analysis, but results in 12% higher potential call locations (MLTA reports 60% higher doing equivalent)

Real World example - CFI

- Tested with an extension based off KCFI sanitizer - security tool that enforces CFG at runtime
- Checks if destination location is valid prior to jumping
- Had a 4.1% runtime overhead on average when compared to a system without this extension



Real World example - Trace/Bug Detection

- Device Allocators clean resources after device is no longer used
  - Cleanup can happen in multiple locations, hard to detect with indirect calls
  - Can result in double frees, use after free, && other incorrect behavior

- Use TFA to identify potential callers in a location, reducing set of functions from n=64 to n=6 when comparing with pure type analysis.

- Compared code that ran after device cleanup callback & the provided callback function

- 8 real bugs in Linux kernel found with this method out of 243 device allocator cleanup functions analyzed

CRITIQUE

Methodological Limitations of TFA

- TFA struggles with unnamed structs and unions in C/C++
  - LLVM **generates unique names** for **unnamed structs** across modules
  - Falls back to imprecise **one-layer type** matching

- Handling virtual calls using static techniques
  - Resource intensive; requires **cataloging all VTables** from class constructors
  - TFA is a static analysis; forced to **attempt precise pointer analysis**
  - **Cannot** fully resolve **runtime polymorphism**

- High Resource Requirements → Scalability Concerns
  - *Liu et al. (2024) used a Linux server (Ubuntu 20.04.1) with **126GB RAM** and an **Intel Xeon Silver 4316 CPU (80 cores)***

Questions Left Unanswered

- Lack of overhead and runtime comparisons to baseline methods
  - How does TFA's overhead expense and runtime efficiency compare to baseline static icall analysis methods?
- Data-flow in highly dynamic behavior
  - How does TFA's virtual call handling compared to dynamic analysis methods?
- Testing done only on C++
  - How would TFA scale to another language? Python? Java?
- Security implications of resolved call targets
  - How secure are the indirect call targets that TFA resolves? How does TFA handle malicious code?

Future Direction

- Hybrid static-dynamic analysis for virtual calls
  - Integrating **dynamic analysis** with **TFA's static approach**
  - **Increased precision** without costing the soundness of the control-flow graph
  - Improve virtual function call resolution

- Scaling TFA to other programming languages
  - Extending TFA to support **multi-language** codebases (e.g., **Python**, **Java**)
  - Test TFA's **generalizability** and **effectiveness** in analyzing various programming environments beyond **C++**

- Extending TFA to improve code security
  - Extending TFA to handle **security checks** and resolve indirect calls in malicious code
  - Ensuring accurate identification of **unsafe or malicious targets**, preventing exploitation in insecure code