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

Dinghao Liu and Shouling Ji, *Zhejiang University;*
Kangjie Lu, *University of Minnesota;* Qinming He, *Zhejiang University*

https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dinghao-improving

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

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

Dinghao Liu[1], Shouling Ji[1], Kangjie Lu[2], and Qinming He[1]

[1]Zhejiang University, [2]University of Minnesota

E-mails: {dinghao.liu, sji}@zju.edu.cn, kjlu@umn.edu, hqm@zju.edu.cn

Abstract

Indirect function calls are widely used in building system software like OS kernels for their high flexibility and performance. Statically resolving indirect-call targets has been known to be a hard problem, which is a fundamental requirement for various program analysis and protection tasks. The state-of-the-art techniques, which use type analysis, are still imprecise. In this paper, we present a new approach, `TFA`, that precisely identifies indirect-call targets. The intuition behind `TFA` is that type-based analysis and data-flow analysis are inherently complementary in resolving indirect-call targets. `TFA` incorporates a co-analysis system that makes the best use of both type information and data-flow information. The co-analysis keeps refining the global call graph iteratively, allowing us to achieve an optimal indirect call analysis. We have implemented `TFA` in LLVM and evaluated it against five famous large-scale programs. The experimental results show that `TFA` eliminates additional 24% to 59% of indirect-call targets compared with the state-of-the-art approaches, without introducing new false negatives. With the precise indirect-call analysis, we further developed a strengthened fine-grained forward-edge control-flow integrity scheme and applied it to the Linux kernel. We have also used the refined indirect-call analysis results in bug detection, where we found 8 deep bugs in the Linux kernel. As a generic technique, the precise indirect-call analysis of `TFA` can also benefit other applications such as compiler optimization and software debloating.

1 Introduction

One of the powerful features of advanced programming languages like C/C++ is the *function pointer*, which allows developers to dynamically invoke different functions depending on the runtime context. Function pointers are widely used in modern software such as OS kernels to implement flexible and efficient functionalities. The invocation site of a function pointer is called an *indirect-call* (*icall* for short in this paper). However, the dynamic nature of icall makes it challenging to determine its possible targets. This challenge hinders the construction of a precise Control-Flow Graph (CFG), which is essential for many significant program analysis and protection tasks (e.g., bug detection, program optimization, and control-flow integrity).

Researchers have been actively proposing techniques to identify icall targets. There are two main categories of techniques for this problem. The first category is dynamic techniques, which record the icall targets at runtime and update the control flow graph accordingly [7, 23, 27, 51, 53]. However, it has limitations in terms of soundness and performance, as it depends on the code coverage and the runtime support. Therefore, we focus on the second category: static techniques, which compute the possible icall targets without executing the program. Some static techniques rely on whole program pointer analysis to identify icall targets [20, 24, 47, 54], but this approach faces challenges in balancing efficiency, soundness, and effectiveness [29, 42]. In particular, precise interprocedural pointer analysis is often infeasible for analyzing large programs such as OS kernels (e.g., Andersen [21]). Moreover, the pointer analysis itself presupposes a global control flow graph, which implies knowing the icall targets beforehand.

As a result, most existing practical solutions rely on type analysis for identifying icall targets [40, 50, 58, 60, 63, 64]. Specifically, they consider all *address-taken functions* (i.e., functions whose addresses have been referenced) that have the same function type as the icall as possible targets. This is known as *type matching* or *signature matching*. This approach is common in CFI schemes [10, 48, 50] and static program analysis [28, 60, 63]. However, it suffers from high false positives, as many unrelated functions can also match the function type of the icall, especially for simple types (e.g., `void (*)(int)`). To address this limitation, *two-layer type matching* [37, 43, 68] is proposed based on the observation that most icalls in large programs are stored and used in com-

Qinming He and Shouling Ji are the co-corresponding authors.

posite types like structures. By using the outer-layer struct type information as an additional filter, it can eliminate many invalid icall targets. `TypeDive` [42] extends this idea and proposes *multi-layer type matching* (*MLTA*), where it tracks the outer-layer struct types recursively (i.e., one struct type is stored into another struct type) to obtain richer type information for icall analysis.

The state-of-the-art method, MLTA, has achieved great success in precisely identifying icall targets, and has been adopted in many static analysis tasks [22, 32, 39, 62, 69]. However, it still suffers from several problems. (1) It is ineffective for programs that rarely use composite types. For instance, we found that only 25% of icalls in the OpenSSL library involve composite types, and thus most icalls in OpenSSL cannot benefit from MLTA. (2) Even for large programs that extensively use composite types, the impact of icalls that cannot benefit from MLTA (i.e., icalls that do not involve composite types) is highly significant. Our experimental results indicate that the minority (23%) icalls that do not involve composite types account for the majority of icall targets (77%) in the Linux kernel.

One fundamental problem of existing approaches is that they do not fully utilize type and data-flow information. In this paper, we present a new approach, `TFA` (Type and Flow co-Analysis), for precise and sound indirect-call resolution. The basic idea of `TFA` is to leverage the complementary advantages of type analysis and data-flow analysis to resolve indirect-call targets optimally. Specifically, for icalls with rich type information, type analysis (e.g., MLTA) could provide us an initial CFG, which enables the inter-procedural data-flow analysis. For icalls with limited or general type information (i.e., MLTA is ineffective), data-flow analysis kicks in to help eliminate unrelated targets. Given an initial CFG generated by MLTA, `TFA` iteratively improves it through type and data-flow co-analysis until convergence is reached. The refined CFG will make the subsequent analysis round more precise and more manageable because it narrows the scope of inter-procedural analysis.

While the intuition of `TFA` is straightforward, implementing `TFA` needs to overcome two challenges. (1) The type information used for icall analysis is not always accurate. Many type-based icall analysis methods [37, 42, 43] are implemented based on LLVM [13], a popular static analysis framework. However, we discovered a series of LLVM composite types that lack essential information (§2.2). We call them *broken types* in this paper. Broken types affect the precision and soundness of type-based analysis and introduce both false positives and false negatives. (2) Directly performing inter-procedural data-flow analysis to track the data-flow relations between icalls and their targets is costly, especially for large programs like OS kernels. Furthermore, `TFA` needs to repeat the analysis iteratively, which poses a challenge to the efficiency of `TFA`.

To address the first challenge, we propose a type recovery technique that leverages the source code information to identify and reconstruct the missing or broken types. Our technique could enhance not only our icall analysis, but also general type-based analysis in LLVM. To address the second challenge, we present a two-dimensional data-flow analysis method customized for icall analysis. It captures both the data-flow dependencies between icalls and their targets, and the type information that is hidden behind complex data-flows. The rich type information obtained by our data-flow analysis significantly improves the existing type analysis methods.

We have implemented `TFA` based on LLVM and evaluated `TFA` with five representative C/C++ programs: the *Linux kernel*, the *FreeBSD kernel*, the *OpenSSL library*, the *OpenCV library*, and the *MongoDB database*. `TFA` could analyze the complex Linux kernel in less than two hours. The experimental results show that `TFA` further eliminates 24% to 59% icall targets compared to the state-of-the-art multi-layer type analysis method. The evaluation also indicates that the type recovery method is effective in improving the soundness of type analysis. The data-flow analysis of `TFA` does not introduce any additional false negatives. We demonstrate the practical benefits of `TFA` in two applications: CFI and bug detection. We have implemented a prototype of a forward-edge CFI scheme leveraging `TFA` for the Linux kernel, where each icall's valid target set is computed according to its unique semantic. We also use `TFA` to detect bugs related to the misuse of device release call-back functions in the Linux kernel. We confirm 6 double-free bugs and 2 memory leak bugs that are hidden behind indirect-calls and would be missed by existing static bug detection tools. In summary, we make the following contributions:

- **A new approach for icall target analysis.** We propose a new approach, `TFA`, to precisely identify icall targets. `TFA` integrates the type analysis and data-flow analysis in such a way that they complement each other to compute icall targets optimally. The approach is designed to be practical for large programs, both kernel and user levels. We will open source `TFA` to facilitate further researches.

- **New techniques.** We propose multiple techniques to address challenges in implementing `TFA`. We propose a type recovery technique to fix the broken types in LLVM. We also design a customized two-dimensional data-flow analysis for icall target resolving, which not only precisely finds icall targets for cases where type analysis fails, but also identifies richer type information.

- **Extensive evaluation and feasible applications.** We extensively evaluate `TFA`'s efficiency, effectiveness, and soundness on five large-scale programs. Compared with the state-of-the-art methods, `TFA` further eliminates 24% to 59% icall targets without introducing more false negatives. The evaluation results confirm that `TFA`'s icall analysis is precise, scalable, and sound. We then implement a strengthened

```
1  typedef void (*f_ptr)(int a, int b);
2  struct S {f_ptr field1; f_ptr field2};
3
4  void address_taken_func1(int a, int b){...}
5  void address_taken_func2(int a, int b){...}
6  void address_taken_func3(int a, int b){...}
7  void address_taken_func4(int a, int b){...}
8
9  struct S s1 = {.field1 = address_taken_func1,
10                .field2 = address_taken_func2};
11 struct S s2 = {.field1 = address_taken_func3,
12                .field2 = address_taken_func4};
13
14 void main() {
15     ...
16     s1.field1(100, 200); // address_taken_func1 is called here
17     ...
18 }
```

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

forward-edge CFI scheme based on our precise icall analysis results and apply it to the complex Linux kernel. We also apply TFA in bug detection and found 8 new bugs in the Linux kernel.

2 Background and Motivation

2.1 Type-Based Indirect-Call Analysis

We present an example of an icall in Figure 1 to compare different type analysis methods. This icall occurs at line 16 and has a function type defined at line 1. A one-layer type matching method would consider any address-taken function with the same function type as the icall as a possible target. Thus, it would include all four functions defined from line 4 to line 7 as valid targets. A two-layer type matching method would also examine if an address-taken function is assigned to a field of a structure, which is a common pattern in large software such as OS kernels. In this example, the icall is retrieved from the first field (`.field1`) of struct `S`. Only `address_taken_func1()` and `address_taken_func3()` are used to initialize this field. Therefore, two-layer type matching identifies these two functions as valid targets of the icall, which eliminates 50% of false positives compared to one-layer type matching.

2.2 Problems with the Type Analysis in LLVM

LLVM is one of the most widely used static analysis frameworks, which is essential for implementing various type-based icall analysis techniques. However, LLVM suffers from two issues that compromise the soundness of type analysis. In this paper, we call these issues *broken types*.

2.2.1 Unreliable Type Equality Checking

One essential requirement of the aforementioned type-based analysis is to determine the equality of different types, especially for struct types. However, even such a fundamental

```
1  /* source code */
2  struct A {
3      int i;
4      int (*f)(int, struct A*);
5      int (*g)(char, struct A*);
6  };
7
8  /*Expected LLVM IR of struct A */
9  %struct.A = type {i32, i32 (i32, %struct.A*), i32 (i8, %struct.A*)}
10
11 /*Actual LLVM IR of struct A */
12 %struct.A = type {i32, {}*, i32 (i8, %struct.A*)}
```

Figure 2: Example of omitting function pointer fields.

```
1  /* source code */
2  struct dvb_usb_adapter_properties adapter[2];
3
4  /*Expected LLVM type of variable adapter */
5  [2 x %struct.dvb_usb_adapter_properties]
6
7  /*Actual LLVM type of variable adapter */
8  <{{i8, i8, i32 (%struct.dvb_usb_adapter*, i32)*,
9  i32 (%struct.dvb_usb_adapter*, i32, i16, i32)*, {i8, i8, i8,
10 {%struct.anon.163, [8 x i8]}}}, %struct.dvb_usb_adapter_properties}>
```

Figure 3: Example of type unfolding.

task is hard to accomplish in LLVM intermediate representations (IRs). The default type-pointer comparison is only applicable for types within the same LLVM module. Some previous works [42, 43] rely on the hashed type strings to perform cross-module type comparison, which is effective for primitive types (e.g., integers). However, this approach is unsound in comparing struct types due to the following anomalies in LLVM type representations:

- **Omitting function pointer fields.** In some cases, the function pointer field of a structure will be replaced as an empty pointer (`{}*`) rather than its point-to type, as shown in Figure 2. This issue has been reported and discussed in several different technical forums [2–5], but it still persists in LLVM 15. Some developers suggest this is a known bug in Clang [1, 2].

- **Missing struct names.** A common programming practice is to initialize a global variable at the point of its declaration (e.g., line 9 and line 11 in Figure 1). In our analysis on the Linux kernel, we found that 17.8% of global struct variables are declared without specifying their struct type names in LLVM IRs. This poses a challenge for the two-layer type matching algorithm, as we cannot determine that `address_taken_func1()` is used to initialize a field of struct `S` in Figure 1, which occurs for 13.5% of outer-layer struct types in the Linux kernel.

- **Type unfolding.** One challenge in analyzing the content of a struct's fields is that they may be partially unfolded, as shown in Figure 3. Sometimes, an `i64` type could be split into two `i32` types. Based on our manual analysis, this scenario is likely to occur when a structure has a union field.

These type representations make type comparisons based on type strings unreliable. LLVM provides a built-in type comparison method [9], which recursively verifies every struct field of the two struct types. LLVM linker also employs another type comparison method based on isomorphism checking [8]. Denisov et al. proposed a tree-automata-based type comparison method for LLVM [6]. However, none of these methods can guarantee the soundness of type comparison.

2.2.2 Type Information Omission in Optimized Code

One of the key tasks in type-based analysis is to determine the type information of different layers (i.e., identify which field of a structure a variable belongs to). In LLVM IRs, this is done by using the `getelementptr` instructions, which are used to access struct fields. For instance, given a GEP instruction like *%ptr = getelementptr inbounds %struct.S* %0, i64 0, i32 5*, we can infer that the pointer `%ptr` originates from the fifth field of struct S.

Previous works on icall analysis have focused on programs compiled with no optimization (O0) [42, 43], and the impact of code optimization on type analysis has not been well studied. We discovered that many GEP instructions in optimized code (e.g., O2 optimization level), which is often preferred by static analysis techniques for its advantages on data-flow analysis [52, 60, 69], would lose crucial type information. For example, the GEP instruction above would be transformed into *%ptr = getelementptr inbounds i8, i8* %0, i64 28* after optimization. The struct pointer type is replaced by a primitive pointer type `i8*`. The field index is replaced by a byte offset based on the primitive pointer. In this scenario, existing methods are unable to retrieve the nested struct types or fields.

2.2.3 Causes of Broken Types

We argue that the above abnormal cases should not be regarded as mere bugs, since they do not cause any observable runtime errors in our experiments. The optimization that eliminates type information in §2.2.2 is justified, as it simplifies the address calculation and improves the runtime performance. Previous work also shows that types in LLVM IRs are not always consistent with the source code (i.e., cases in §2.2.1) for the sake of *optimal machine code generation* [36]. However, this poses a challenge for researchers who want to perform type-based analysis on LLVM IRs. Therefore, instead of modifying LLVM's type system directly, we propose a program analysis oriented type recovery approach to address the issues caused by broken types.

2.3 Type Recovery and Co-Analysis

As discussed in §2.2, broken types can exhibit various patterns, but they all share a common problem: the lack of type information. To address this challenge, we present a novel type recovery technique (§4.1) that can infer the missing type information for broken types, enabling more sound and precise type-based analyses (e.g., icall analysis).

The icall in Figure 1 has only one possible target: `address_taken_func1()`, which cannot be determined by type analysis alone. However, if we can leverage additional data-flow analysis to identify that the icall only involves variable `s1`, we can reduce the set of potential targets to `address_taken_func1()` and `address_taken_func2()`. By combining the results of two-layer type matching and data-flow analysis, we can achieve the exact ground-truth result in this example. This observation inspires us to design and implement `TFA`.

3 Overview

The goal of `TFA` is to resolve icall targets soundly and precisely. The key idea of `TFA` is to combine multi-layer type analysis and data-flow analysis in an iterative manner to refine the control-flow graph (CFG) progressively. The overview of `TFA` is shown in Figure 4, which consists of two analysis phases.

In the first phase, `TFA` takes the LLVM IR files as inputs and performs type recovery on them. `TFA` scans all struct variables and GEP instructions of the program in the LLVM IRs to detect and fix broken types. Specifically, `TFA` leverages the source code information and recovers the missing type information layer by layer. Based on the recovered type information, `TFA` then conducts multi-layer type analysis to generate an initial CFG of the program, which enables inter-procedural data-flow analysis in the next phase. `TFA` also records the detailed analysis results of each icall in this phase to guide the subsequent data-flow analysis.

In the second phase, `TFA` refines the icall targets iteratively using a *two-dimensional data-flow analysis*. This analysis establishes the data-flow relationships between the icalls and their possible targets, and also extracts layered type information from inter-procedural context. These two analysis flows enhance the icall analysis directly and indirectly, respectively.

The icall target analysis produces a refined CFG in each round, and stops when the CFGs of two consecutive rounds are sufficiently similar. The final CFG is then stored in a database to facilitate further static analysis.

4 System Design

In this section, we first introduce our type recovery approach, which aims to improve existing type analysis. Next, we present the customized alias analysis method we used. Finally, we introduce the two-dimensional data-flow analysis of `TFA`, which cooperates with the type analysis on refining icall targets.

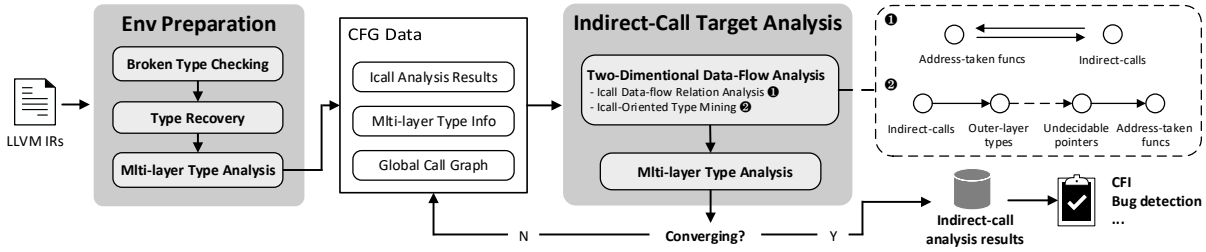

Figure 4: The overview of TFA. It has a preparation phase and an iterative icall analysis phase. It takes the LLVM IRs as inputs and outputs the icall analysis results to a local database.

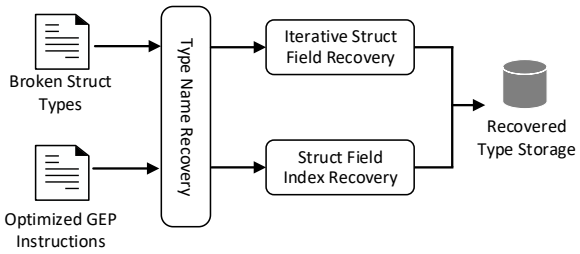

Figure 5: The overview of type recovery.

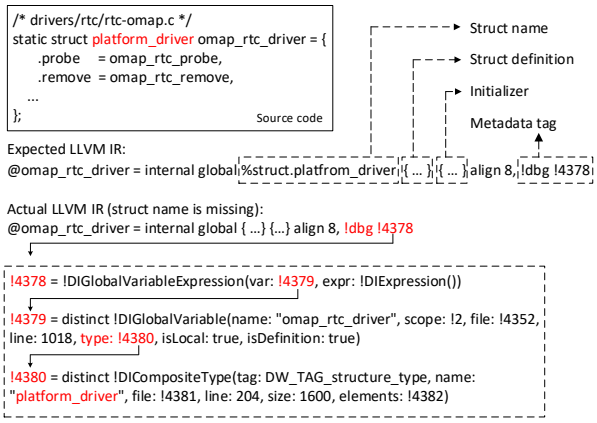

Figure 6: The workflow of type name recovery.

4.1 Type Recovery

Type recovery aims to recover the missing struct type information to support precise and sound type analysis. The overview of type recovery is shown in Figure 5. It takes two types of inputs that have incomplete or distorted type information: (1) the broken struct types (i.e., cases in §2.2.1), and (2) the optimized LLVM GEP instructions (i.e., cases in §2.2.2). For the broken struct types, TFA will reconstruct the erased struct type names and fields. For the optimized GEP instructions, TFA will infer the struct types and field indices from the optimized types (i.e., `i8*`) and byte offsets.

For both types of inputs, the first step of type recovery is to recover the corresponding struct type names. For the broken struct types, we then recursively examine and restore their possible broken field types. For the optimized GEP instructions, we additionally recover the field indices that the instructions originally accessed. TFA maintains a global map to store the recovered types for facilitating subsequent type-based static analysis in LLVM.

4.1.1 Type Name Recovery

We present a case study from the Linux kernel to demonstrate the process of type name recovery, as shown in Figure 6. This case involves a global variable `omap_rtc_driver` with a struct type named `platform_driver`. Ideally, its representation in LLVM IR should include three essential elements: struct name, struct definition, and initializer. However, its struct name is absent. We examined the entire Linux kernel and discovered 64,457 global struct variables (17.8%) that lacked their struct type names in the definition.

Our approach is to leverage the source code information (stored in the LLVM metadata nodes) to recover the missing struct name. For instance, in Figure 6, by tracking the metadata layer by layer (`!4378`→`!4379`→`!4380`), we can identify the type name of `omap_rtc_driver` and construct a complete struct type in LLVM IRs.

For an optimized LLVM GEP instruction, we first obtain the pointer operand of this instruction (e.g., `i8* %0` in the example of §2.2.2). We then inspect the use-chain of this value to locate the `llvm.dbg.value` instruction, which records the corresponding source code information. Finally, we extract the metadata node from this instruction and recover the type name of this value (`Struct.S` in the example of §2.2.2) using the same process as in Figure 6.

4.1.2 Iterative Struct Field Recovery

Struct types that contain fields of other struct types are frequently used in programming. One challenge of the type recovery is that the broken struct types could propagate through nested struct fields, and we need to recover all possible broken

field types accurately. For instance, if the name of a struct type is missing, its nested struct fields are also likely to have missing names. Usually, the incorrect types are mixed with correct types, which complicates the recovery process.

To address this problem, we propose the following recovery methods for a given broken struct type. We first recover the missing name with the workflow in §4.1.1. We then search the type definition list of the current module and find the intact struct definition (i.e., *identified struct type*) of the broken type based on the recovered name. Finally, we iteratively match all fields of the broken and intact struct types to recover all potential broken field types. In the matching procedural, we apply three additional strategies to ensure soundness: (1) We recursively unfold all composite types into primitive types to enable sound and effective type comparison. (2) We split all integer types into multiple `i8` types according to their bit-width. (3) We conservatively allow void pointer type to cast to any function pointer type.

4.1.3 Struct Field Index Recovery

For an optimized GEP instruction, we need to additionally recover the exact struct field index that the GEP instruction intends to access. Since we have recovered the original struct type names, we can use the LLVM API [1] to obtain the field index by computing the byte offset. However, this approach is not always reliable. We manually examined the corresponding LLVM IRs and discovered that the code optimization pipline could merge multiple GEP instructions. For instance, given a C code snippet `s = a->b->c`, LLVM would normally generate two GEP instructions to get the address of variable `s`: one GEP for computing the address of `b` from `a`, and another for computing the address of `c` from `b`. However, these two GEP instructions could be combined into one during the code optimization, which sums up the two address offsets and loses the information of struct `b`. Consequently, existing methods fail to analyze the struct accessing behaviors correctly.

To address this problem, we propose an algorithm to identify the types and indices for the merged fields, as illustrated in Algorithm 1. The basic idea of our algorithm is to recursively compute the address offset layer by layer until it matches the given offset. In particular, when we encounter a struct field (line 5), we find the nearest field index according to the current offset (line 6) and compute the delta between this field's offset and the given offset (line 7-8). If the delta is non-zero, then the GEP must access a subfield of the current struct field, and we use the delta as the new offset to locate the subfield. When we encounter a struct array field (line 12-14), we simply descend into the element struct type and use the remainder of the given offset and the element type size to locate the subfield (line 15-17).

[1] LLVM API getElementContainingOffset(): given a valid byte offset into the struct type, it returns the struct index that contains it.

Algorithm 1: Get layered types and indices

Input: BST: Base struct type of the GEP instruction;
$offset$: Byte offset in the GEP instruction
Output: $STList$: The recovered struct type list;
$indexList$: The index list of the fields in $STList$

```
1  cur_offset ← 0;
2  CST ← BST;
3  index ← 0;
4  while offset > 0 do
5      if CST is Struct Type then
6          index ← CST.GETINDEXFROMOFFSET(offset);
7          cur_offset ← CST.GETELEMENTOFFSET(index);
8          offset ← offset − cur_offset;
9          STList.PUSH_BACK(CST);
10         indexList.PUSH_BACK(index);
11         CST ← CST.GETELEMENTTYPE(index);
12     else if CST is Array Type then
13         eleTy ← GETELEMENTTYPE(CST);
14         if eleTy is Struct Type then
15             type_size ← GETTYPESIZE(eleTy);
16             CST ← eleTy;
17             offset ← offset mod type_size;
18         end
19     else
20         return NULL, NULL;
21 end
22 return STList, indexList;
```

4.2 Alias Analysis

To perform data-flow analysis of TFA, we need to track alias relationships among program variables. We adopt a data structure called *alias graph* [35, 38] to store these relationships. The alias analysis method is inter-procedural, flow- and field-sensitive, and it achieves a good trade-off between efficiency and precision for large-scale programs. In this paper, we enhance the soundness of the alias analysis for icall analysis scenario by making the following modifications:

Flow-insensitivity. Flow-sensitivity is a property of alias analysis that takes control-flow information into account. For our analysis, we require both forward and backward data-flow exploring, which makes the flow-sensitivity irrelevant. Therefore, we use a flow-insensitive approach that simplifies the computation and improves the soundness of our analysis.

Field-insensitivity. Field-sensitivity is the ability to distinguish different fields of the same composite object. We do not support field-sensitivity in alias analysis for two reasons: (1) Our analysis result is the intersection of data-flow analysis and multi-layer type analysis, which already provides field-sensitivity. (2) The Linux kernel has a mechanism, called `container_of`, to obtain the starting address of a struct value from its field value. This mechanism introduces soundness issue [41] because it prevents us from getting the correct outer-layer struct type (which is missing) or the correct index

of the current field (which is negative).

Supporting May-Alias Query. Current design of the alias graph based alias analysis mainly supports must-alias query [35], which will introduce soundness issues in icall analysis. Therefore, we re-design the graph updating algorithms to support may-alias query.

We have reimplemented the alias analysis according to the above requirements. Our analysis is integrated with a data-flow analysis framework that performs alias and data-flow analysis simultaneously. Whenever a new instruction is processed by the data-flow analysis, the alias analysis updates the global alias graph accordingly. We provide more details about the alias graph structure and the graph updating algorithm in §A.1 in the Appendix.

4.3 Two-Dimensional Data-Flow Analysis

Previous work has tried to utilize data-flow analysis or point-to analysis to resolve icall targets [30, 49, 55, 61]. These techniques attempt to establish the data-flow or alias relationships between icalls and address-taken functions directly. However, for large-scale programs (e.g., OS kernels), these relationships can be very complex and intricate, which leads to inefficiency, imprecision, and unsoundness of existing data-flow based analysis methods [29, 42].

Our key observation is that existing multi-layer type analysis can effectively reduce the false positives of icall analysis by exploiting the rich layered struct type information, but such information is often obscured by complex data-flows. Therefore, improving icall analysis does not necessarily require a complete data-flow analysis between icalls and their targets, but a lightweight type information mining. Based on this insight, we present a two-dimensional data-flow analysis to improve icall analysis. Specifically, it aims to analyze not only the direct data-flow relationships between icalls and their targets, but also icall-related layered struct type information across inter-procedural contexts.

4.3.1 Icall Data-Flow Relation Analysis

Bidirectional Data-Flow Analysis. To obtain rich data-flow information, `TFA` analyzes the data-flows of icall traces inter-procedurally from two directions, i.e., *bidirectional data-flow analysis*. (1) A backward data-flow analysis that starts from an icall and identifies the possible address-taken functions that can be its targets. After completing the backward analysis, `TFA` calculates the intersection of the alias set of the icall and the result of multi-layer type analysis to obtain the final icall targets. (2) A forward data-flow analysis that starts from an address-taken function and determines which icalls can have it as one of their targets. For icalls that are not in the alias set of the address-taken function, `TFA` can safely exclude the address-taken function from their target sets.

Fall Back Strategies. Occasionally, data-flow analysis may confront cases that compromise analysis soundness and result in false negatives. To cope with this problem, we have introduced several early termination conditions designed to preserve soundness while balancing effectiveness and efficiency. The data-flow analysis halts and the results are conservatively reverted to type analysis upon triggering any of the specified conditions:

- **The number of the aliased value reaches the threshold.** `TFA` allows users to specify a maximum number (500 by default) of values that are aliased with the input value (e.g., an icall pointer). `TFA` terminates the data-flow analysis when the number of aliased values reaches the limit and falls back to type analysis.
- **The input value is aliased with assembly code.** Assembly code is currently out of the scope of our analysis. Therefore, when the input value is aliased with assembly calls, `TFA` terminates the data-flow analysis and falls back to type analysis. In addition, `TFA` also stops the data-flow analysis when the subgraph includes global variables named `llvm.compiler.used` (`llvm.used` in LLVM 14 and earlier versions). Usually, this could happen when the program initializes and updates global variables through assembly code (e.g., Linux static calls).
- **The input value is aliased with arithmetic operated values.** A pointer value may be modified by arithmetic operations in practice. Static analysis of its alias set is still an open challenge, and the soundness of the results is hard to guarantee. Therefore, `TFA` terminates the data-flow exploration in this case and falls back to type analysis.

4.3.2 Icall-Oriented Type Mining

The state-of-the-art multi-layer type analysis achieves more precise icall analysis by utilizing richer layered struct type information. However, in large programs (e.g., OS kernels), the storage relationships between structural types and icalls could be complex (e.g., involves inter-procedural data-flows). As a result, existing methods fail to resolve these cases, which limits the effectiveness of MLTA. `TFA` aims to resolve this problem through data-flow based type mining. However, type mining can be costly and impractical for programs with a large number of structural types. To address this problem, we present *icall-oriented type mining*, a technique that focuses on tracking and extracting type information that is relevant to indirect call analysis from two perspectives.

Type Mining for Icalls. We found that many indirect function calls with nested type information are not handled correctly in multi-layer type analysis. A common case in the Linux kernel is illustrated in Figure 7. Multi-layer type analysis assumes that the indirect call at line 4 lacks any enclosing struct type information and resorts to one-layer type analysis.

```
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.

However, there is a caller of `interleaved_copy()` at line 16, which passes a function pointer `transfer` as an argument. The origin of `transfer` reveals that it does have an enclosing type `substream->ops->copy_kernel` at line 16.

To make these icalls benefit from multi-layer type analysis, we perform a backward data-flow analysis starting from them and track their origins. `TFA` conducts an inter-procedural analysis to determine if an icall is derived from a struct field, which indicates aliasing with a struct field. Unlike the previous data-flow analysis, this phase terminates early when an aliased value with nested type information (i.e., GEP instructions in LLVM IRs) is encountered during data-flow propagation.

Type Mining for Undecidable Targets. Multi-layer type analysis requires recording the confined targets for a struct type. However, this could be infeasible when a struct field has an undecidable target (i.e., *type-escaping* [42]). Figure 8 illustrates a typical case in the Linux kernel. A function pointer is assigned to the `detach` field of the struct pointer `ap` at line 7, which has the type `struct drm_aperture`. Since the function pointer originates from the argument (`detach` at line 3), its potential identities cannot be determined by type analysis alone. Multi-layer type analysis maintains a global map to record the fields of structs that encounter type-escaping (e.g., `drm_aperture->detach` in this case). When an indirect call has an escaped outer-layer type (e.g., the indirect call at line 16), multi-layer type analysis will fall back to one-layer type matching conservatively to ensure soundness.

`TFA` addresses this problem through backward data-flow tracking. In this example, we choose the function pointer whose source is undecidable (i.e., `detach` at line 3) as the starting point of data-flow analysis. `TFA` precisely identifies the origin of the input pointer (`drm_aperture_detach_firmware()` at line 22). This allows us to eliminate the escaped layered types (`drm_aperture->detach`) from the global escaping map. As a result, more icalls could benefit from MLTA.

Mutual Enhancement. Our type mining approach for indirect calls has a remarkable feature: its two analysis components can mutually enhance each other in the iterative data-flow analysis process. The type mining for indirect calls enables more indirect calls to benefit from the multi-level type analysis, but it also generates more undecidable targets. The type mining for undecidable targets effectively addresses this issue. This forms a positive feedback loop that improves the precision of indirect call resolution.

```
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 }
```

Figure 8: Example of type-escaping because of undecidable targets.

5 TFA Implementation

We have implemented `TFA` on LLVM of version 15-init as several analysis passes, which contains 12k lines of C++ code (including 1.7k LoC of the implementation [2] of `TypeDive` [42] for multi-layer type analysis). It accepts unlinked LLVM bitcode files as inputs, and outputs the analysis results into a local MYSQL database.

5.1 LLVM Bitcode Generation

We use an independent LLVM pass for bitcode generation, which uses LLVM API `WriteBitcodeToFile()` to dump bitcode files. We modify the Makefiles of our target programs to load this pass. We add `-g` flag into the CFLAGS of Makefiles to retain debug information for type recovery and icall information storage. Additionally, we have observed instances where the same struct type is compiled as a *packed struct* in certain modules and as a normal struct elsewhere, resulting in disparate struct layouts that affect the type recovery in `TFA`. To resolve this inconsistency, we employ the Clang compilation option `-mms-bitfields` to ensure uniform struct layouts across different modules.

[2] https://github.com/umnsec/mlta

5.2 Type Comparison

In TFA, we use struct type names to compare different struct types. For struct types with the same name in different modules, we include the struct's definition file as part of the struct name. The missed type names would be recovered through type recovery. However, developers could define struct or union types without names in C/C++. LLVM will automatically generate type names for them (e.g., `%union.anon.157`, `%struct.anon.8`). The type recovery of TFA conservatively regards them as invalid because different LLVM modules could generate different names for the same nameless struct field. This affects the type analysis when a function pointer is stored in a nameless field. In such cases, we fall back to one-layer type matching, which means that any icall that has the same function type as the function pointer can potentially target it. Fortunately, this limitation only affects 270 (0.4%) address-taken functions in the Linux kernel.

5.3 Multi-Layer Type Analysis

We adopt the implementation of existing state-of-the-art multi-layer type analysis as a part of TFA with necessary modifications. Specifically, we use the implementation [3] of TypeDive [42] with the following modifications: (1) We set the outer-layer number as two, so TypeDive executes two-layer type matching for icall analysis. Considering that we need to iteratively execute multi-layer type analysis, two-layer type matching is the most cost-effective type analysis for us. (2) We rewrite its type recording system to apply our type recovery approach.

5.4 Kernel Macro Optimization

The Linux kernel macro `EXPORT_SYMBOL` is frequently utilized to make functions from one kernel module available in other modules. Theoretically, it should not affect the analysis of icalls. However, the function pointer associated with this macro emits an LLVM IR pattern identical to that of assembly code, encompassing `llvm.used` for data-flow propagation and potentially causing premature termination of the analysis. To mitigate this issue, we meticulously examine the source code of each address-taken function by checking the corresponding LLVM metadata. Should a function be engaged with `EXPORT_SYMBOL`, TFA continues the data-flow propagation without interruption.

5.5 Supporting C++

In C++ programs, virtual function calls constitute a significant proportion of icalls. According to our manual analysis, such icalls are often devoid of sufficient outer layer type information. Furthermore, the analysis of data-flows associated with these icalls tends to be resource-intensive. Consequently, existing approaches for type or data-flow analysis tend to be ineffective for such icalls. To address this challenge, we employ the strategy outlined in TypeDive [42]. Specifically, we analyze the constructors of all C++ classes to catalog their virtual function tables (VTables). Upon the invocation of a virtual function call, we first identify the class issuing the call and subsequently pinpoint the precise callee function by referencing the corresponding VTable.

[3] https://github.com/umnsec/mlta

5.6 Convergence Checking

Previous works [30, 42, 68] evaluate the effectiveness of icall analysis based on the number of icall targets. In this paper, we adopt the same metric to compare the effectiveness of different analysis rounds of TFA. We consider the analysis to have reached convergence and terminate the analysis when the difference of icall targets between adjacent analysis rounds is less than 10,000.

6 Evaluation

We aim to answer the following research questions (RQs) in our evaluation.

- **RQ1:**Compared to existing methods, how effective and efficient is TFA in refining icall targets?
- **RQ2:** Could TFA guarantee the soundness after introducing data-flow analysis?
- **RQ3:** How do broken types affect icall analysis? Moreover, how effective is the type recovery in addressing this challenge?
- **RQ4:** Does TFA benefit real-world applications?

6.1 Experimental Setup

We evaluate TFA on top of a Linux server (Ubuntu 20.04.1) with 126GB RAM and an Intel Xeon Silver 4316 CPU at 2.30GHz (80 cores). We choose five widely used programs as the analysis targets: the Linux kernel (v5.18), the FreeBSD kernel (v12.4), the OpenSSL library (v3.0.6), the OpenCV library (v4.9.0), and MongoDB (v8.0.0). Additionally, for the Linux kernel, we built it with two different configurations: the `allyesconfig` to evaluate the scalability of TFA, and the `localmodeconfig` to emulate the most common kernel usage scenario.

6.2 Performance on Eliminating Icall Targets

TFA aims to eliminate false positives for icall analysis and provide a more fine-grained icall target result. Similar to existing

icall analysis works [30, 42, 68], we use the average icall target number to evaluate the performance of TFA on eliminating icall targets. We leverage the OpenMP library [15] to enable multi-core parallel acceleration for TFA, with a concurrency of 32 threads in our evaluation. Even for the Linux kernel, which comprises nearly 28 million lines of code, TFA could complete the entire icall analysis in two hours. This result is promising, given the complexity of the analysis task.

6.2.1 Performance Comparison with Existing Tools

Comparison with Type Analysis. We present the comparison of three icall target analysis methods in Table 1: signature (the most commonly used method), MLTA (the state-of-the-art method), and TFA. MLTA effectively reduces false positives compared with the signature matching method. In MLTA's paper [42], only icalls that have multi-layer type information are included in the evaluation. Therefore, it reports a higher precision. In comparison, our evaluation includes all icalls. The results indicate that TFA could further eliminate 24% to 59% of icall targets compared with state-of-the-art type-based MLTA method.

For C++ programs, multi-layer type analysis is suboptimal in eliminating redundant icall targets. The virtual function analysis method presented in §5.5 demonstrates superior performance. Even when both virtual function analysis and MLTA are applied, our system, TFA, succeeds in eliminating 44% to 56% of icall targets in C++ programs, thereby confirming its effectiveness.

Comparison with Program Modularization. We further compare TFA with TyPM [41], a program modularization based icall analysis approach. The basic idea of TyPM is to refine icall analysis by performing type matching locally within modules that have type dependencies, rather than globally as in existing techniques. We evaluate TyPM on the Linux kernel compiled with the `localmodeconfig` and measure its effectiveness in reducing icall targets. TyPM further eliminates 19.8% of icall targets compared with the results of MLTA. On comparison, TFA eliminates additional 55.4% of icall targets compared with MLTA for the same program, demonstrating the benefits of type and data-flow co-analysis of TFA.

Comparison with Abstract Interpretation. Abstract interpretation is a technique that projects a program's behavior into an abstract domain, systematically over-approximating the possible states of the program, and by extension, the potential targets of icalls. We investigated two representative tools: Frama-C [18] and Ikos [25]. Frama-C actually necessitates developers to explicitly annotate potential icall targets via specialized comments, which implies that precise and automated icall analysis remains unaddressed by this tool. Furthermore, an attempt to apply Ikos to the Linux kernel did not culminate within 12 hours, and a significant portion of the analysis was marred by substantial runtime errors arising from unsupported LLVM types, affecting approximately 40% of the modules. These observations indicate that for extensive codebases, abstract interpretation-based methods for icall analysis may lack scalability. TFA, in contrast, aims to provide a more feasible solution.

Comparison with Other Co-Analysis Method. KELP [26] shares a similar observation with TFA: a multitude of function pointers in icall analysis are straightforward and could be resolved through affordable data-flow analysis. KELP conducts localized pointer analysis for such functions and integrates this analysis with MLTA. As the source code of KELP was unavailable at the time of this paper's writing, and its icall analysis metrics appear to differ from those used by TFA, a high-level comparison is provided here for clarity and completeness. The primary distinctions in design between KELP and TFA are twofold: (1) KELP makes data-flow analysis as a separate preliminary stage prior to MLTA, whereas TFA integrates data-flow analysis and type analysis into a unified co-analysis framework, allowing for mutual enhancement. (2) KELP is designed for a single execution with an emphasis on efficiency, whereas TFA employs an iterative analysis to achieve greater accuracy.

6.2.2 Performance of Analysis Rounds and Phases

In this subsection, we conduct an in-depth evaluation of the performance across different analysis rounds within our two-dimensional data-flow analysis, as well as its distinct phases. Due to page constraints, we have selected the Linux kernel (kernel space C program), OpenSSL (user space C program), and MongoDB (user space C++ program) as representative targets for evaluation.

Analysis Rounds. Table 2 shows the numbers of the total icall targets that remain after each analysis round of TFA. The two-dimensional data-flow analysis utilizes the results of MLTA as its initial inputs. For MongoDB, the initial results incorporate virtual function analysis subsequent to MLTA. We found that the first analysis round eliminates the most icall targets. The convergence speed of our analysis depends on the size and complexity of the target software. Smaller software could reach convergence faster than larger software. However, even for OS kernel-level software, our analysis converges within three rounds. This demonstrates the effectiveness and efficiency of TFA for resolving indirect calls in real-world software.

Analysis Phases. We studied the impact of each analysis phase in our two-dimensional data-flow analysis, as presented in Table 3. The bidirectional data-flow analysis has the most significant effects on eliminating the number of icall targets for all the three programs. The two components of icall-oriented type mining also contribute substantially to the precision of our analysis.

Table 1: Icall target analysis results. Avg. indicates the average icall target number. Sig, MLTA, and MLTA+VH indicate the signature matching, multi-layer type analysis, and multi-layer type analysis + virtual function analysis, respectively.

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

Table 2: Results of different analysis rounds of TFA. Each number in the table indicates the number of total icall targets after the current round of co-analysis analysis.

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	-

Table 3: Results of different analysis phases of two-dimensional data-flow analysis. The BDA in the table indicates the bidirectional data-flow analysis. The TM-I and TM-UT indicate type mining for icalls and undecidable targets, respectively. Each number in the table indicates the reduced number of total icall targets brought by corresponding analysis phases.

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

6.3 False Negative Analysis

One important concern for TFA is to ensure the soundness of the data-flow analysis results. In this subsection, we present the false negative analysis of TFA and the method we used to collect icall traces as the ground-truth for evaluation. We selected the Linux kernel and OpenSSL library as the evaluation targets in this section.

6.3.1 Trace Collection

We collect icall traces through LLVM instrumentation for the Linux kernel. Specifically, we compile the kernel with Clang and insert a hook function that records the icall information before each icall site. This function logs the source location and the callee name of each icall to the system log. We implement the function in the kernel source code and make it globally visible. It utilizes the kernel API `sprint_symbol()` to retrieve the callee names from their addresses. We also filter out any duplicate traces before logging them to simplify the subsequent trace analysis. To obtain a sufficient number of icall traces, we run the Linux Test Project [12] on the Linux kernel. We finally collect 6,929 unique traces. Our instrumentation is implemented as an LLVM pass with 230 lines of C++ code.

The tracing strategy of OpenSSL is similar to the OS kernels, where we also instrument the program to collect traces. To obtain the name of the callee function from the symbol table, we initially attempted to use the Linux API `dladdr()`, but this API often failed and returned empty results. Hence, we leveraged the *LLVM prefix data* to identify callees. In particular, we assigned a fixed-size function name as the prefix data for every address-taken function, which could be accessed with a fixed offset from the function's entry point. We compiled the hook function as a static library and linked it to OpenSSL by modifying its Makefile. We stored the icall traces in a local MYSQL database. We used the *openssl speed* [16] benchmark to collect icall traces and obtained 851 unique traces in total.

6.3.2 Results for False Negative Analysis.

We apply two criteria to select the icall traces that are valid for our analysis. First, we discard traces that do not have valid callee names, which indicates that the symbols are not resolved correctly. Second, we exclude traces that have mismatched icall or callee locations with the source code, which implies that the binary code is not consistent with the source code. After applying these filters, we obtain 6,452 unique and valid icall traces from the Linux kernel and 683 traces from the OpenSSL library.

FN Results of the Linux Kernel. TFA only misses two callees in our analysis, which come from two icalls in function `__apply_relocate_add()`. Specifically, these two icalls miss the same callee: `__memcpy()`. The reason for the two false negatives is that our *type analysis* phase fails to identify the callee. We examine the whole kernel code and discover that the address of `__memcpy()` is never assigned to any pointer (i.e., it is not an address-taken function). There are only a few direct calls to this function. Therefore, `__memcpy()` is

not considered as a possible icall target by our type analysis, even when we use signature matching. We also investigate the icall target sets produced by TFA, and we notice a wrapper function of `__memcpy()`: `memcpy()`. We suspect that some compiler optimizations replace `memcpy()` with `__memcpy()` and cause this false negative.

FN Results of the OpenSSL Library. We identified 58 false negatives in OpenSSL in total. We investigate these cases and find all of them are missing since *type analysis* (signature matching). In particular, the OpenSSL tends to define icalls with general parameter types (e.g., `void *`). However, the actual parameters could cast to other types (e.g., `MD5_CTX *`). This is a common challenge for many existing icall analysis works [30, 42, 58], and leads to false negatives even for signature matching. One feasible solution for this problem is conservatively equating void pointers with any other pointer types. We applied this approach to TFA and reevaluated its performance on the OpenSSL library. The average number of icall targets for both signature matching and MLTA increased significantly (63.6%↑ and 60.2%↑). However, the impact on TFA was relatively small (12.1%↑). Moreover, this approach eliminated all false negatives for TFA.

> ***Finding 1:*** The data-flow analysis of TFA does not introduce more false negatives than existing type-based analysis methods. TFA performs better when we adopt more conservative type analysis methods.

6.4 Effectiveness of Type Recovery

To better understand how broken types impact type-based analysis, in this subsection, we use the traces collected from the Linux kernel to check the effectiveness of our type recovery.

6.4.1 Broken Struct Type Recovery

To evaluate the impact of broken struct types in §2.2.1, we build a Linux kernel with O0 optimization and turn off the type recovery feature in TFA. We then reexamine the icall analysis outcomes. The number of false negatives for the Linux kernel rises to 879 (13.6%). Figure 9 illustrates a typical false negative case in the Linux kernel, where the callee `con_write_room()` is omitted for the icall at line 4. The omitted callee is assigned to the `write_room` field of struct type `tty_operations` at line 13. Ideally, when the icall is generated from the same field of the same struct type (i.e., the `ops->write_room` at line 4), the callee should have been detected as a possible target. However, the representations of the struct type `tty_operations` are inconsistent in the two LLVM bitcode files, where one of them represents two function pointers' types as `{}*`. Consequently, type analysis considers them as distinct struct types, which results in false negatives. By

```
1  /* drivers/tty/tty_ioctl.c */
2  unsigned int tty_write_room(struct tty_struct *tty){
3      if (tty->ops->write_room)
4          return tty->ops->write_room(tty);
5      return 2048
6  }
7
8  /* drivers/tty/vt/vt.c */
9  static unsigned int con_write_room(struct tty_struct *tty){...}
10
11 static const struct tty_operations con_ops = {
12     ...
13     .write_room = con_write_room,
14     ...
15 };
16
17 // struct tty_operations in tty_ioctl.bc
18 %struct tty_ioctl_operations = type {... {}*, {}*, ...}
19
20 // struct tty_operations in vt.bc
21 %struct tty_ioctl_operations = type {... i32 (%struct.tty_struct*)*,
22 i32 (%struct.tty_struct*)*, ...}
```

Figure 9: False negative example of MLTA in the Linux kernel.

applying type recovery, we can eliminate all false negatives caused by broken struct types.

6.4.2 Optimized GEP Instruction Recovery

To evaluate the impact of optimized GEP instructions, we build a Linux kernel with O2 optimization and disable the GEP instruction recovery in TFA. We repeat the icall analysis results and observe that the number of false negatives rises to 80. The main sources of false negatives are imprecise storage analysis of address-taken functions and nested struct fields, which require accurate identification of nested struct types and field indices from GEP instructions. By applying type recovery, we successfully eliminate 77 false negatives. There is one additional false negative compared to the previous experiment due to a GEP instruction that performs an implicit type casting. Since this scenario is rare, we leave it as future work.

> ***Finding 2:*** The broken types significantly influence the soundness of type analysis. The type recovery in TFA effectively prevents false negatives introduced by broken types.

6.5 Application I: Fine-Grained CFI

We implemented a forward-edge CFI scheme that leverages our fine-grained icall analysis results, which is built based on the KCFI sanitizer [10, 11]. It prevents control-flow hijacking attacks by disallowing changes to the pre-defined CFG. Currently, our CFI scheme is implemented on LLVM IRs in Clang compiler to support various backends. We load the valid target sets from the database during compilation and encode them as read-only data. We utilize LLVM prefix data to determine which function will be called before we emit the icall. We insert a verification function before each icall to check if the callee belongs to the corresponding target set.

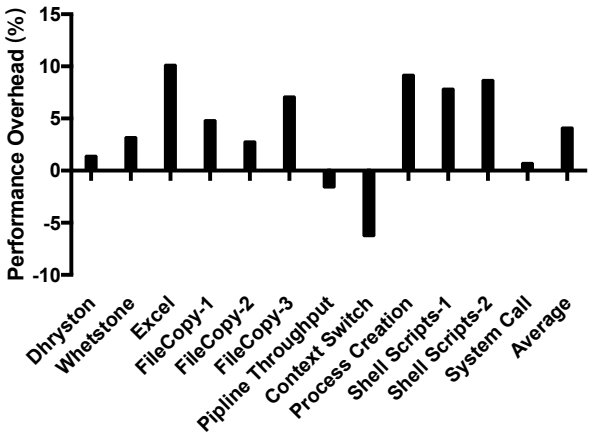

Figure 10: Overheads of the CFI scheme.

If an invalid function is invoked, the CFI scheme redirects the control flow to a user-defined error-handling function for further debugging. Our scheme is compatible with any forward-edge CFI scheme that allows specifying a target set for each independent icall (e.g., Clang CFI [58]).

We measured the performance of our forward-edge CFI scheme on a Ubuntu virtual machine with 4 CPUs (Intel Core i7-8770 CPU, 3.20Ghz). We used the UnixBench 5.1.2 [17] as the benchmark, which offers a diverse range of workloads and is widely used in CFI evaluation [30, 37]. Compared with the original system without CFI protection, the average system performance decreased only by 4.1% on average. The detailed overheads of each task are shown in Figure 10. The evaluation result confirms that our fine-grained CFI scheme is feasible in practice. Notably, we observed negative overheads, a phenomenon also reported in other CFI research [30]. This improvement may be attributed to the CFI instrumentation altering the code layout, resulting in enhanced code alignment or more efficient utilization of the instruction cache, thereby accelerating context switches. The overhead of the CFI could be further optimized through architecture-specific (e.g., x86) implementation, which is mainly an engineering effort. We leave this part as further work.

6.6 Application II: Bug Detection

One of the potential applications of icall analysis is static bug detection, which requires a precise CFG for inter-procedural analysis. We leverage our fine-grained icall analysis to analyze the Linux kernel, focusing on a specific type of icalls: the release call-back functions in devices. These functions are responsible for resource cleanup when a device is no longer in use. They are controlled by reference counting and assigned by the device allocators. However, the device allocators may also have their own error handling code for resource cleanup, which could conflict with the release call-back functions. This scenario involves three challenging mechanisms for static analysis: indirect-call, reference counting, and error handling. Existing methods typically overlook this scenario and only address one of these mechanisms [33, 34, 43, 46, 56, 57, 65].

TFA Assisted Trace Analysis. To understand the triggering mechanism of the release callback, we perform a trace analysis in the first step. We use the icall analysis results generated by TFA and trace backward from a callback function. The trace consists of six function calls originating from `put_device()`, among which three are indirect calls. TFA accurately determines that the three indirect calls have only one caller each. With TFA's assistance, a non-expert researcher can complete this task in 15 minutes. If we use type analysis instead of TFA's co-analysis for icall target identification, there would be 64 possible callers of the release callback, and we would have to manually examine all these callers and their transitive callers.

Bug Detection. From the trace analysis, we identify that the release callback is invoked by `put_device()` after the device reference count reaches zero. In this step, we use TFA to detect two types of bugs in the release callbacks: (1) redundant cleanup, which may lead to use-after-free or double-free vulnerabilities, and (2) missing resource release, which may cause memory leaks. We compared the release callback code with the error handling code after `put_device()` in the device allocators to detect these bugs. Out of 243 release callbacks generated by co-analysis, TFA reported nine bugs. We manually verified them and found eight real bugs, including six double-free bugs and two memory-leak bugs, as shown in Table 4. Three of them have been fixed by other developers in the latest kernel. We submitted patches for the remaining bugs and four of them have been applied. If we use type analysis instead of co-analysis, we would have 671 potential release callbacks, and 63.8% of them would be false positives, which would make the bug detection ineffective.

Table 4: List of bugs detected in the Linux kernel. The S, A, and F in the Status column indicate submitted, applied, and fixed by other developers in the latest version, respetively.

Bug function	Impact	Status
zfcp_port_enqueue	Double-free	A
ptp_ocp_device_init	Double-free	A
ocxl_file_register_afu	Double-free	F
rpmsg_virtio_add_ctrl_dev	Double-free	F
rpmsg_probe	Double-free	F
stm_register_device	Double-free	S
css_alloc_subchannel	Memleak	A
i3c_master_register_new_i3c_devs	Memleak	A

7 Discussion

Assembly Code Analysis. At present, TFA does not have the capability to analyze assembly code. When encountering assembly code, we adopt a conservative approach and terminate

the data-flow analysis, falling back to type analysis. Prior works [30, 37] rely on manual analysis to resolve indirect calls and jumps in assembly code. We manually checked some failure cases due to assembly calls in the Linux kernel and found many of them are introduced by a new feature called *Linux static call*. We plan to design a dedicated analysis pass for this feature in our future work.

Type Analysis in LLVM. In addition to the broken types discussed in §2.2, prevalent type analysis methods, such as signature matching, also falter due to type casting. For instance, unexpected type casting can engender substantial false negatives in the analysis of icalls within the OpenSSL library. To address this problem, Ge et al. used taint analysis to infer icall targets [30]. IFCC [58] used two more conservative approaches to construct icall target set: *Single* (which collects all functions into a single set) and *Arity* (which infers icall targets according to the number of arguments). FINE-CFI [37] presented some IR-based methods to address this problem, but they were ineffective for eliminating the false negatives in our analysis. As a result, LLVM 16 introduces an *opaque pointer type* to replace all pointer types containing their point-to types, due to various issues with the latter [14]. Furthermore, typed pointers have been deprecated in LLVM 17+, as the LLVM community discourages the reliance on LLVM's internal type system for conducting type-based analyses. Nevertheless, the latest LLVM releases have introduced 'tbaa' metadata [19], designed to represent the type system of higher-level languages and aid in type analysis. This metadata has been employed in the implementation of type-based alias analysis. We posit that this feature holds considerable promise as a solution, and we propose the examination of this new capability as an avenue for future work.

8 Related Work

Indirect-Call Analysis for CFI Schemes. CFI schemes usually give priority to the soundness of icall analysis. Many existing CFI schemes adopt Single or Arity to construct icall targets [58, 59, 66, 67]. These methods do not rely on type information and can be applied to binary executables. A more precise approach is to use the types of function pointers to match icall targets [10, 48, 50], but this approach suffers from soundness degradation due to primitive type casting. Recently, some CFI schemes have leveraged the idea of MLTA to further refine icall targets [30, 37]. However, they do not address the problem of broken types, which can affect the security of CFI. In this paper, we systematically study the causes and impacts of this problem and propose a type recovery system to solve it.

Indirect-Call Analysis for Bug Detection. Icall analysis is essential for static bug detection, which relies on a global CFG to perform inter-procedural analysis. A common challenge for this task is the high false positive rate caused by inaccurate icall analysis. Therefore, some tools opt to ignore icalls altogether [33, 38, 45]. Many static bug detection tools adopt one-layer type matching to resolve icalls (e.g., Deadline [63], LRSan [60], and K-MELD [28]). When multi-layer type analysis was introduced, it was quickly applied to bug detection [22, 32, 39, 52, 62]. Recently, *directed fuzzing* also demands precise icall analysis to calculate the cross-function distance [44]. We believe `TFA` could benefit both existing and future static or dynamic analysis.

Type and Data-Flow Co-Analysis. Ghavamnia et al. proposed two filtering schemes to refine the results of Andersen's points-to analysis and obtain more accurate icall targets [31]. These schemes exactly perform one-layer type matching, which do not exploit the rich layered type information for optimization. FINE-CFI [37] introduced the concept of *struct location vector* to enable two-layer type analysis. However, when an icall originates from a function argument, FINE-CFI falls back to one-layer type matching. The data-flow analysis approach of `TFA` could effectively resolve this problem. Ge et al. applied taint analysis from address-taken functions for icall analysis [30]. When a function pointer is assigned to a struct's field, all memory objects of this struct's field are tainted, which is similar to two-layer type matching. This method relies on two strict assumptions on function pointer operations to support its taint analysis. When a violation occurs, users have to interrupt the analysis and manually resolve it. `TFA` does not make any assumptions or require frequent manual intervention. `TFA` also improves the icall analysis through mining the hidden layered types, which is missed by existing methods.

9 Conclusion

In this paper, we present `TFA`, which performs type and data-flow co-analysis to optimally resolve indirect-call targets. We also propose several techniques to enhance the scalability, soundness, and precision of `TFA`. We evaluate `TFA` on five famous C/C++ programs, and show that `TFA` could further eliminate 24% to 59% of indirect-call targets compared with the state-of-the-art approaches. We also apply `TFA` to improve the security of forward-edge CFI and the ability of static bug detection. As a generic technique, we believe that the precise indirect-call analysis of `TFA` can benefit various security research domains.

10 Acknowledgment

We sincerely appreciate our shepherd and all the anonymous reviewers for their insightful comments on our work. This work was supported by the Key R&D Program of Zhejiang Province (2022C01086). Kangjie Lu was supported in part by the NSF awards CNS2045478, CNS-2106771, CNS-2154989, and CNS-2247434. Any opinions, findings, conclusions or

recommendations expressed in this material are those of the author and do not necessarily reflect the views of NSF.

References

[1] 2013. Bug 14920: incomplete conversion of recursive (function) types. https://bugs.llvm.org/show_bug.cgi?id=14920

[2] 2013. Why a function pointer field in a LLVM IR struct is replaced by {}*? https://stackoverflow.com/questions/18730620

[3] 2016. Function pointer type becomes empty struct. https://lists.llvm.org/pipermail/cfe-dev/2016-November/051601.html

[4] 2016. Function pointer type becomes empty struct. https://lists.llvm.org/pipermail/cfe-dev/2016-November/051633.html

[5] 2016. Function pointer type becomes empty struct. https://lists.llvm.org/pipermail/cfe-dev/2016-November/051635.html

[6] 2020. Type Equality in LLVM. https://lowlevelbits.org/type-equality-in-llvm/

[7] 2022. CFGgrind. https://github.com/rimsa/CFGgrind

[8] 2022. LLVM function areTypesIsomorphism(). https://www.llvm.org/docs/doxygen/IRMover_8cpp_source.html

[9] 2022. LLVM function cmpTypes(). https://llvm.org/doxygen/FunctionComparator_8cpp_source.html

[10] 2023. KCFI sanitizer. https://reviews.llvm.org/D119296

[11] 2023. KCFI Support. https://lwn.net/Articles/893164/

[12] 2023. Linux Test Project. https://github.com/linux-test-project/ltp

[13] 2023. The LLVM Compiler Infrastructure. https://llvm.org

[14] 2023. Opaque Pointers in LLVM. https://llvm.org/docs/OpaquePointers.html

[15] 2023. The OpenMP API specification for parallel programming. https://www.openmp.org

[16] 2023. OpenSSL Speed. https://www.openssl.org/docs/man1.1.1/man1/openssl-speed.html

[17] 2023. UnixBench. https://github.com/kdlucas/byte-unixbench

[18] 2024. Frama-C - Framework for Modular Analysis of C programs. https://www.frama-c.com

[19] 2024. LLVM 'tbaa' Metadata. https://llvm.org/docs/LangRef.html#tbaa-metadata

[20] Martn Abadi and Mihai Budiu. 2005. lfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Vol. 1. 2.

[21] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. Citeseer.

[22] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe DMA Accesses in Device Drivers. In *30th USENIX Security Symposium (USENIX Security 21)*. 1629–1645.

[23] Mohamad Barbar, Yulei Sui, Hongyu Zhang, Shiping Chen, and Jingling Xue. 2018. Live path cfi against control flow hijacking attacks. In *Australasian Conference on Information Security and Privacy*. Springer, 768–779.

[24] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 353–362.

[25] Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings 12*. Springer, 271–277.

[26] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information. In *33nd USENIX Security Symposium (USENIX Security 24)*.

[27] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *26th USENIX Security Symposium (USENIX Security 17)*. 131–148.

[28] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *The 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*.

[29] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 28–39.

[30] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 179–194.

[31] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1749–1766.

[32] HyungSeok Han, Andrew Wesie, and Brian Pak. 2021. Precise and Scalable Detection of Use-after-Compacting-Garbage-Collection Bugs. In *30th USENIX Security Symposium (USENIX Security 21)*. 2059–2074.

[33] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically detecting error handling bugs using error specifications. In *25th USENIX Security Symposium (USENIX Security 16)*. 345–362.

[34] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. 2019. Detecting error-handling bugs without error specification input. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 213–225.

[35] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. 2018. An efficient data structure for must-alias analysis. In *Proceedings of the 27th International Conference on Compiler Construction*. 48–58.

[36] Lukáš Korenčik. 2019 [cit. 2022-11-19]. *Decompiling Binaries into LLVM IR Using McSema and Dyninst [online]*. Master's thesis. Masaryk University, Faculty of InformaticsBrno. SUPERVISOR: RNDr. Petr Ročkai, Ph.D..

[37] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. 2018. Fine-cfi: fine-grained control-flow integrity for operating system kernels. *IEEE Transactions on Information Forensics and Security* 13, 6 (2018), 1535–1550.

[38] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–872.

[39] Changming Liu, Yaohui Chen, and Long Lu. 2021. KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel.. In *NDSS*.

[40] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1627–1644.

[41] Kangjie Lu. 2023. Practical Program Modularization with Type-Based Dependence Analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1256–1270.

[42] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1867–1881.

[43] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[44] Changhua Luo, Wei Meng, and Penghui Li. 2022. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1050–1064.

[45] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. 2022. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2096–2113.

[46] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 531–544.

[47] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2004. Precise call graphs for C programs with function pointers. *Automated Software Engineering* 11, 1 (2004), 7–26.

[48] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. DROP THE ROP fine-grained control-flow integrity for the Linux kernel. *Black Hat Asia* (2017).

[49] Ben Niu and Gang Tan. 2013. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 199–210.

[50] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 577–587.

[51] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 914–926.

[52] Aditya Pakki and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 1203–1218.

[53] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750.

[54] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886.

[55] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 460–473.

[56] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. 2021. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In *30th USENIX Security Symposium (USENIX Security 21)*. 2471–2488.

[57] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 752–762.

[58] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX security symposium (USENIX security 14)*. 941–955.

[59] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.

[60] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[61] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE symposium on security and privacy*. IEEE, 380–395.

[62] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and detecting disordered error handling with precise function pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. 2041–2058.

[63] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 661–678.

[64] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 221–232.

[65] Dongyang Zhan, Xiangzhan Yu, Hongli Zhang, and Lin Ye. 2022. ErrHunter: Detecting Error-Handling Bugs in the Linux Kernel Through Systematic Static Analysis. *IEEE Transactions on Software Engineering* 49, 2 (2022), 684–698.

[66] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[67] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security* (Washington, D.C.) *(SEC'13)*. USENIX Association, USA, 337–352.

[68] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. PeX: A Permission Check Analysis Framework for Linux Kernel.. In *28th USENIX Security Symposium (USENIX Security 19)*. 1205–1220.

[69] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. 2022. Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3253–3267.

A Appendix

A.1 Customized Alias Analysis

A.1.1 Alias Representation

Figure 11 shows an example of alias graph, in which we define four variables: `a`, `b`, `c`, and `d`. The alias graph of this code snippet consists of three nodes. Each node represents an *alias set*, whose member pointers are aliased with each other. An edge in the alias graph represents a pointer dereference operation. In this paper, the single alias graph is always linear, as shown in the example. Multiple subgraphs constitute the global alias graph that represents the alias relationships for a program.

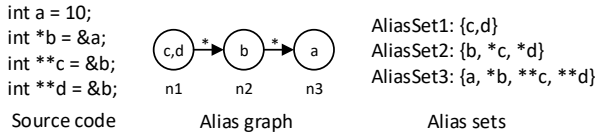

Figure 11: Alias graph and alias set.

A.1.2 Graph Building and Updating

The alias analysis mainly focuses on three types of alias relationships: `Move` (i.e., $v_1 = v_2$), `Load` (i.e., $v_1 = *v_2$), and `Store` (i.e., $*v_2 = v_1$). Figure 12 shows the graph updating process while handing these alias relationships.

Handle Move. The `Handle_Move()` accepts three parameters: the two values v_1 and v_2 with move relationship ($v_1 = v_2$), and the global alias graph `G`. The algorithm first gets the two nodes which v_1 and v_2 belong to respectively, namely n_1 and n_2 (line 1 and line 2). If v_1 or v_2 does not belong to any node in `G`, `GetNode()` will create a new node for it and add it into `G`. It then merges these two nodes into one that contains all values of the original two nodes (line 3). If n_1 or n_2 has a predecessor or successor node, `MergeNode()` will recursively execute node merging for the two alias graphs that contain

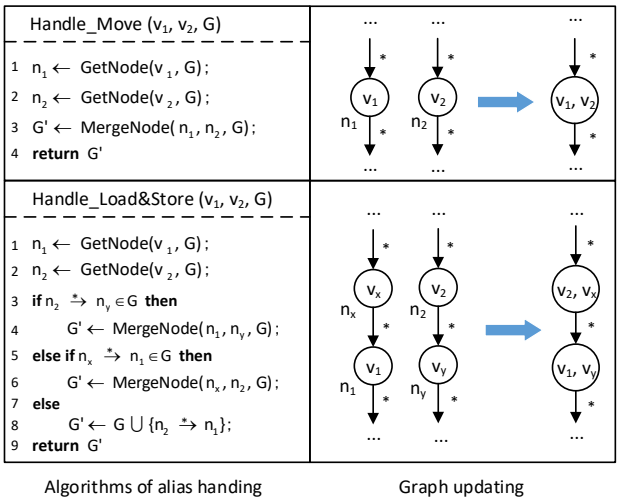

Figure 12: Alias graph building.

n_1 and n_2. The global alias graph will be updated to `G'` after `Handle_Move()`.

Handle Load & Store. The logic for handling `Load` ($v_1 = *v_2$) and `Store` ($*v_2 = v_1$) is the same, where the algorithm also takes two values (v_1 and v_2) and a global alias graph `G` as inputs. It then gets the two nodes that v_1 and v_2 belong to (i.e., n_1 and n_2) and checks whether there exists an edge that starts from n_2 and ends at another node (e.g., n_y) in `G` (line 3). If so, the algorithm will execute a recursive node merging the same way as handling `Move` for n_1 and n_y (line 4). If not, the algorithm then checks whether there exists an edge ending at n_1 and starting from another node (e.g., n_x) in `G` (line 5). If the edge exists, the algorithm recursively merges n_x and n_2 (line 6). If none of the above-mentioned cases happen, we only need to add a new edge starting from n_2 and ending at n_1 into `G` (line 8).