DEEP\textsc{Type}: Refining Indirect Call Targets with Strong Multi-layer Type Analysis

Tianrou Xia\hfill Hong Hu\hfill Dinghao Wu
Pennsylvania State University\hfill Pennsylvania State University\hfill Pennsylvania State University

Abstract

Indirect calls, while facilitating dynamic execution characteristics in C and C++ programs, impose challenges on precise construction of the control-flow graphs (CFG). This hinders effective program analyses for bug detection (e.g., fuzzing) and program protection (e.g., control-flow integrity). Solutions using data-tracking and type-based analysis are proposed for identifying indirect call targets, but are either time-consuming or imprecise for obtaining the analysis results. Multi-layer type analysis (MLTA), as the state-of-the-art approach, upgrades type-based analysis by leveraging multi-layer type hierarchy, but their solution to dealing with the information flow between multi-layer types introduces false positives. In this paper, we propose strong multi-layer type analysis (SMLTA) and implement the prototype, DEEP\textsc{Type}, to further refine indirect call targets. It adopts a robust solution to record and retrieve type information, avoiding information loss and enhancing accuracy. We evaluate DEEP\textsc{Type} on Linux kernel, 5 web servers, and 14 user applications. Compared to TypeDive, the prototype of MLTA, DEEP\textsc{Type} is able to narrow down the scope of indirect call targets by 43.11% on average across most benchmarks and reduce runtime overhead by 5.45% to 72.95%, which demonstrates the effectiveness, efficiency and applicability of SMLTA.

1 Introduction

Indirect call, used commonly to determine the functions to be called at runtime, is a fundamental feature of C/C++ for achieving dynamic program characteristics. Production software (e.g., nginx) and operating systems (e.g., Linux) intensively utilize indirect calls to dynamically adapt program behaviors according to runtime environments and demands, through loading and linking the desired shared libraries.

Precise identification of indirect call targets is of paramount importance, as the control-flow transitions between indirect calls and their respective targets play an essential role in the construction of a global control-flow graph (CFG), which is extensively adopted in various security-related fields. Static analysis tools rely on CFG for bug detection and program hardening [17, 20, 22, 25, 43, 61–63, 66, 68], program partitioning and privilege separation [5, 8, 18, 26, 28, 42], pruning redundant paths in symbolic execution [4, 7, 10, 50, 58], and guiding directed fuzzing for specific objectives [3, 9, 32, 36, 38, 39, 47]. Additionally, control-flow Integrity (CFI) defenses [1, 6, 14, 34, 35, 49, 65, 67] have been proposed to mitigate control-flow hijacking attacks. However, the strength of CFI depends on the precision of CFG. Imprecise CFG construction results in advanced attacks that bypass CFI [19, 29, 41, 44, 45, 60].

The key challenge in construction of an accurate CFG is identifying the targets of indirect calls. Modern compilers like GCC and Clang cannot determine these targets without additional analysis and instrumentation. Conservative approaches [65, 67] consider all functions or those with address taken as potential targets for each indirect call, producing a considerable number of false positive edges within CFG, which impair the functionality of the applications built upon CFG and impose unnecessary cost. Data tracking analysis [14, 23, 46, 48, 68] tracks value flow, which pursues accuracy at the cost of high performance overhead while the accuracy depends on the precision of the taint analysis and points-to analysis techniques they employ. Type-based analysis [34, 49, 52] checks function signatures to identify functions whose types match with an indirect call. While this approach is efficient, it is susceptible to false positive targets if many functions share the same type as the actual target.

Multi-layer type analysis (MLTA) [27] was proposed for the purpose of improving the accuracy of type-based analysis. Given the fact that function pointers can be members of composite data structures, the type of a function pointer along with the composite types holding it compose a multi-layer type. For example, if function pointer \texttt{ptr} with type \texttt{void \(\text{int}^*\)} is a member of object \texttt{a} with type \texttt{struct A}, the variable \texttt{a.ptr} has multi-layer type \texttt{void \(\text{int}^*\) | struct A}. A function \texttt{f} also has this multi-layer type if it is assigned to \texttt{a.ptr}. MLTA matches multi-layer types of functions and indirect calls to refine indirect call targets.
However, multi-layer types introduce challenges in type matching because address-taken functions may be propagated between different multi-layer types through information flow, making it hard to collect all targets for an indirect call (see §2.2). MLTA bypasses the challenges by splitting a multi-layer type into several two-layer types, and adopts each layer’s type as the basic unit for information storage and type matching (see §2.4), which avoids missing targets while producing false positive targets. It relinquishes the type information between spitted layers and weakens the restrictions provided by multi-layer types, thereby negatively affecting accuracy.

This paper proposes an advanced approach, Strong Multi-Layer Type Analysis (SMLTA), to mitigate the false positive targets produced by MLTA. It adheres to the strong restriction that identifies only those functions as targets whose entire multi-layer types match with the indirect calls. SMLTA addresses the challenges in multi-layer type matching by resolving the relationships between multi-layer types based on the directions of information flow, and utilizes an adapted breadth-first search (BFS) algorithm [56] to discover all multi-layer types engaged in the propagation of target functions. It also employs a conservative strategy to deal with ambiguous type information due to information flow.

We implemented SMLTA within a prototype, DEEPType, which contains two working phases: 1) information collection and 2) target identification. In the first phase, DEEPType establishes and maintains the mappings between multi-layer types and their associated functions (§4.1). The collected multi-layer types are organized in a hierarchical manner for the purpose of quick retrieval (§4.2). It also preserves the relationships between multi-layer types as indicated by the directions of information flow, facilitating the tracing of multi-layer types engaged in function propagation (§4.3). In the second phase, DEEPType determines the multi-layer type of each indirect call and discovers all other multi-layer types engaged in target propagation (§5.1). Then, it verifies whether the engaged multi-layer types match with the indirect call to precisely identify associated functions as targets (§5.2). Additionally, DEEPType handles diverse instructions and code patterns in real-world programs to reduce the inaccuracy caused by corner cases.

We evaluated DEEPType on Linux kernel, 5 server programs, and 14 user-level applications. We compared it with TypeDive (i.e., the prototype of MLTA), as MLTA is the-state-of-the-art approach in type-based analysis. The results indicate that DEEPType outperforms TypeDive in the precision of indirect call target identification, reducing the average number of indirect call targets by 43.11% on average across most benchmarks. In terms of performance, DEEPType decreases the runtime overhead by 5.45% to 72.95% and achieves lower memory consumption in all evaluated benchmarks. Additionally, a case study on a real-world CVE shows that SMLTA is more powerful than MLTA in reducing attack surface and preventing exploits.

In summary, this paper makes the following contributions.

- Propose a novel approach called strong multi-layer type analysis (SMLTA) that employs strong restrictions provided by multi-layer types to refine indirect call targets.
- Develop a prototype, DEEPType, which overcomes challenges in multi-layer type matching and utilizes SMLTA to precisely and efficiently identify indirect call targets.
- Evaluate DEEPType with 20 benchmarks and compare it with TypeDive, exhibiting its capability in further refining indirect call targets and reducing both runtime overhead and memory consumption. Additionally, we demonstrate that SMLTA offers a higher level of security.

2 Motivation

This section clarifies the challenges in indirect call target identification using multi-layer types, describes how MLTA and SMLTA address these challenges, and highlights the accuracy improvement achieved by SMLTA through an example.

2.1 Motivating Example

Listing 1 shows a code snippet with buffer overflow vulnerability. The strcpy at line 25 can overwrite the memory location adjacent to buf if the length of msg is larger than MAX_LEN. The adjacent memory location belongs to w_op, which is afterwards assigned to u->uw at line 26. So, the function pointer u->uw->low_priv can point to an address manipulated by the attacker after buffer overflow occurs. At line 29, this function pointer is used to make an indirect call, resulting in arbitrary execution if a lenient CFG is deployed on this program. To prevent such control-flow hijacking attack and other unexpected bugs caused by imprecise CFG, precisely identifying indirect call targets is crucially required.

2.2 Challenges

The use of multi-layer types presents challenges in matching indirect calls with all potential targets, particularly when functions are propagated across multi-layer types due to information flow. We categorize the challenges into three classes consistent to three forms of information flow.

First, type assignment and casting operations can transform one multi-layer type into another, resulting in information flow from the original to the transformed type, as well as affecting members of composite types. For example, the type assignment operation (line 26) indicates information flow from w_op to u->uw, as well as w_op->low_priv to u->uw->low_priv. In this context, the function write_to_shared_mem with multi-layer type void (char*)* | struct.Write (line 7), should be propagated to void (char*)* | struct.Write | struct.User. Because information flow affects
typedef void (*fp)(char*);
struct Write {fp low_priv; fp high_priv;};
struct User {struct Write *uw; ...};
struct Kernel {struct Write *kw; ...};

void func_init(struct Write *w_op, struct Kernel *k) {
    w_op->low_priv = &write_to_shared_mem;
    w_op->high_priv = &write_to_protected_mem;
    k->kw->low_priv = &write_to_protected_mem;
    k->kw->high_priv = &write_to_kernel_mem;
}

void user_priv_write(fp icall_ptr, char *buf) {
    (*icall_ptr)(buf);
}

void write_to_mem (char *msg) {
    struct User *u;
    struct Kernel *k;
    struct Write *w_op;
    char buf[MAX_LEN];
    strcpy(buf, msg); // buffer overflow
    func_init(w_op, k);
    if (low_priv()) (*u->uw->low_priv)(buf);
    if (user_mode()) {
        u->uw = w_op;
        strcpy(buf, msg); // buffer overflow
        if (user_mode()) {
            if (low_priv()) (*u->uw->low_priv)(buf);
            else user_priv_write(u->uw->high_priv, buf);
        }
    }
}

Listing 1: A program vulnerable to control-flow hijacking attack through indirect call. The strcpy at line 25 has buffer overflow vulnerability, which allows attackers to rewrite the function pointer u->uw->low_priv and redirect control-flow through the indirect call at line 29.

In example descriptions, we use simplified function names for readability.

MLTA utilizes the extra type information extracted from composite data structures and supports field-sensitivity considering that a composite data structure may have multiple members holding different functions. If one function is assigned to a pointer, MLTA records the mappings between the multi-layer type and the function, which is called "type-func confinement". It splits multi-layer types and uses each layer type along with an index as key, as Table 1 shows. Similarly, the original and transformed types in type assignment and casting operations are logged in a split manner as well.

Table 1: Mappings between types and functions in MLTA for Listing 1. MLTA splits multi-layer types into two-layer types and maintains mappings between these types and associated functions. The two-layer types are presented by a composite type along with an index, which indicates the member type at a specific position. For example, struct.Write with index 0 represents the two-layer type void (char*)* | struct.Write. The numbers in parenthesis indicate the line numbers where the functions are confined to the types.

In Listing 1, the indirect call at line 29 has type void (char*)*, which matches with functions *shared*, *protected*, and *kernel*. As a result, all of them are identified as targets though only *shared* is the real target. Similarly, the indirect call at line 15 also has the three targets while only function *protected* is the real target.

2.4 MLTA

MLTA utilizes the extra type information extracted from composite data structures and supports field-sensitivity considering that a composite data structure may have multiple members holding different functions. If one function is assigned to a pointer, MLTA records the mappings between the multi-layer type and the function, which is called "type-func confinement". It splits multi-layer types and uses each layer type along with an index as key, as Table 1 shows. Similarly, the original and transformed types in type assignment and casting operations are logged in a split manner as well.

In Listing 1, the indirect call at line 29 has multi-layer type void (char*)* | struct.Write | struct.User. MLTA identifies potential targets by collecting associated functions for each layer and calculating the intersection to find common functions. For the first layer void (char*)* | struct.Write with index 0, both have no original type. Thus, MLTA retrieves the associated functions from Table 1, resulting in set {*shared*, *protected*, *kernel*} for the first layer and set {*shared*, *protected*} for the second layer. The third layer struct.User with index 0 has an original type struct.Write (line 26), thus MLTA gathers the associated functions for both types, generating a set {*shared*, *protected*}.
protected*, *protected*, *kernel*]. Finally, it computes the intersection of these three sets. The common functions in resulting set { *shared*, *protected*, *kernel* } are identified as targets. The indirect call at line 15 with type void (char*)* has no original type. So, MLTA directly generates the target set { *shared*, *protected*, *kernel* }, according to Table 1. In contrast with tradition type-based analysis, MLTA narrows down the target set for the indirect call at line 29, but there is still a visible gap between MLTA result and ground truth.

In MLTA, the first challenge is addressed by splitting multi-layer types. The complex information flow between them is simplified to straightforward flow between individual types, which is easier to track. The second challenge is addressed by conservatively confining a function to each layer of its multi-layer type, ensuring no missing target even if the outer layers are discarded. To overcome the third challenge, MLTA marks all general pointer types and composite types that interact with them as "escaping types". It skips the layers with escaping types when calculating intersection, which removes the impact of general pointer types.

### 2.5 SMLTA

SMLTA employs entire multi-layer types as keys for information storage, as Table 2 shows, to circumvent the false positive targets caused by splitting multi-layer types. The relationships between multi-layer types are recorded in the same way. We call multi-layer type T2 a "friend type" relative to multi-layer type T1 if information flows from T2 to T1 (i.e., a function may be propagated from T2 to T1). To identify targets for an indirect call, SMLTA exhaustively searches for its friend types and gathers all associated functions as targets.

In Listing 1, the indirect call at line 29 has multi-layer type void (char*)* | struct.Write#0 | struct.User#0 where "#0" indicates the index. SMLTA exhaustively discovers all friend types that may have information flowing to this multi-layer type, only void (char*)* | struct.Write#0 in this example, and then gathers associated functions of these multi-layer types. The resulting set { *shared* } contains the identified target, which is exactly the real target. The indirect call at line 15 has type void (char*)*. Because the function pointer icall_ptr is a parameter, SMLTA adds a "fuzzy type" as its outer layer, which matches with any type. Thus, this indirect call can match with all address-taken functions whose first layer is void (char*)*, resulting in the target set { *shared*, *protected*, *kernel* } without missing potential targets.

In SMLTA, the first challenge is addressed by maintaining the relationships between multi-layer types and the exhaustive search of all friend types engaged in function propagation. The second challenge is addressed by fuzzy type, which conservatively admits that all types could possibly match with the discarded outer layer types, ensuring no missing targets. The third challenge is also solved by conducting an exhaustive search of friend types. This search treats general pointer types as bridges between composite types, unblocking function propagation stuck on general pointer types.

### 3 Overview

The prototype of SMLTA is called DEEPType, which contains two phases, as is shown in Figure 1.

Given bitcode file(s) as input, phase 1 analyzes initialization instructions and type propagation instructions to collect information. An initialization instruction assigns a function to a function pointer. We say, it "confines" the function to the corresponding multi-layer type. The mappings between the entire multi-layer types and associated functions are stored in Type-Func Map. For quick access and retrieval purpose, DEEPType archives these multi-layer types in Type Lookup Maps in a hierarchical manner using multi-layer mappings. Type propagation instructions include type assignment and casting instructions that possibly propagate functions from one multi-layer type to another through information flow. DEEPType records the mappings between destination and source multi-layer types of information flow in Type-Type Map using entire multi-layer types as keys and values. The sources are "friend types" relative to destinations.

In phase 2, DEEPType analyzes each indirect call instruction to figure out its multi-layer type. If the function pointer is a formal parameter, fuzzy type is added to represent potentially discarded outer layers. Then, DEEPType exhaustively searches for friend types from Type-Type Map, retrieves multi-layer types that match with the indirect call or its friend types from Type Lookup Maps, and obtains associated functions from Type-Func Map. The union of all associated functions are potential targets for the indirect call. Finally, DEEPType outputs a list of indirect calls in the analyzed program along with their locations and respective targets.

### 4 Phase 1: Information Collection

This section presents how SMLTA collects information from initialization and type propagation instructions, and how it organizes multi-layer types hierarchically.
4.1 Type-Function Confinements

A function could be considered as a potential target of an indirect call only if it is utilized to initialize local and/or global variables. We confine the function to the multi-layer type of the initialized function pointer by establishing a mapping between them in Type-Func Map. It is straightforward to determine the multi-layer types of local variables, by extracting the types layer by layer as they are loaded. However, the multi-layer types of formal parameters and nested global variables may not be fully ascertainable using the same method.

The multi-layer type of a formal parameter is uncertain because it may have extra outer layers that are lost during parameter passing. To complete its multi-layer, we introduce fuzzy type, as defined in Definition 1, to cover for missing layers. The fuzzy type can be matched with any type. For instance, void (int)* | fuzzy type matches with any multi-layer type whose first layer is void (int)*, including void (int)* | void (int)* | struct.A#0, and void (int)* | struct.A#0 | struct.X#0, etc. Similarly, when an index is uncertain, we employ fuzzy index to conservatively match with any index.

Definition 1 Fuzzy type marks the type of an uncertain layer. The existence of this layer is uncertain, and the type of this layer is uncertain. In type verification, a fuzzy type can match with any type, even if the corresponding layer does not exist.

A nested global variable is one that initializes members of other variables, thus may have extra layers involved in other initialization instructions. Its multi-layer type is uncertain when analyzing the instruction initializing it. To gather complete type information, we track all variables that hold this global variable iteratively. For instance, Listing 2 shows a nested global variable x86_64_elf32_vec (line 8) with type struct bfd_target, which initializes a member of another global variable _bfd_target_vector (line 13). Thus, the function pointer _bfd_read_ar_hdr_fn (line 4), as a member of the global variable, should be confined to both void (bfd*)* | struct bfd_target#53 and void (bfd*)* | struct bfd_target#53 | vector bfd_target#237, ensuring that a potential target could be identified through both multi-layer types.

4.2 Multi-layer Type Organization

For efficient access and retrieval, we organize the collected multi-layer types hierarchically using multi-layer mappings and store them in Type Lookup Maps. We prepare N maps for a program, in which the multi-layer types have up to N+1 layers. More details are illustrated in Appendix A.

To archiver a multi-layer type T, the first n layers of T is used as a key in the nth map (where 1 ≤ n ≤ N), and the nth layer of T is stored as the corresponding value. If T consists of M layers, where M is less than N, only the first M maps are utilized for storage. Figure 2 shows an outline of Type Lookup Maps and exemplifies the utilization of this data structure via two multi-layer types: ① void (int)* | struct.A | struct.X and ② void (int)* | struct.B | struct.Y | struct.P.

The First Map stores the mapping between the common first-layer type void (int)* and respective second-layer types struct.A, for ①, and struct.B, for ②. The Second Map stores mappings between first-two-layer types and corresponding third-layer types, among which the first entry is for ① and the second entry is for ②. The Third Map works in a similar way to store the rest part of ②. If a multi-layer type contains more than four layers, we will establish more maps based the number of layers, to record the rest parts.

---

3In LLVM IR, index is typically derived from GetElementPtrInst when the corresponding operand is a constant. However, index may be non-constant operand such as a phi instruction, indicating an uncertain index until runtime.

4The indexes are omitted to simplify the example and emphasize the key point, which is how to hierarchically archive multi-layer types.
4.3 Type Relationship Resolving

A function can be propagated from one multi-layer type to another via type propagation instructions. To collect all involved multi-layer types, we analyze type assignment and casting instructions and introduce friend type, as defined in Definition 2, to describe their relationships.

Definition 2 A is a friend type relative to B if either of the following holds:

1. There exists information flow from A to B.
2. There exists information flow from A to C, and B to C.

Due to the information flow, the associated functions of A are propagated to B. We say, A "shares" functions with B.

The second condition manifests that one multi-layer type is a friend type relative to another if there exists a chain of information flow following the first condition. This chain can be as long as the entire program. We record the mapping between one multi-layer type and its direct friend types, satisfying the first condition, in Type-Type Map. The indirect friend types, satisfying the second condition, can be inferred by the recorded relationships.

A type assignment instruction assigns the value of one variable to another. They have different multi-layer types, but share the same first-layer type. Such instructions create a one-way relationship: the source type of information flow is a friend type relative to the destination type. For example, Listing 3 shows a type assignment operation (line 10) where the return value of function bfd_get_section_by_name is assigned to pupdate->section, resulting in information flow from the source type asection to the destination type asection | struct.section_add#5, which allows asection to share its associated functions with asection | struct.section_add#5.

A type casting instruction transforms one multi-layer type into another, with the first layers of each being distinct. In Listing 4, the original type of entry is struct.bfd_hash_entry* (line 2). It is transformed to struct.string_hash_entry at line 4. The type casting at line 6 transforms struct.string_hash_entry to struct.bfd_hash_entry*. Considering the flow-insensitivity feature of our static analysis, it is uncertain whether a type casting instruction executes before or after an indirect call, making it uncertain whether the multi-layer type of the indirect call is transformed from another multi-layer type. To address this ambiguity, we conservatively establish a bidirectional relationship between the source and destination types, allowing them to share associated functions with each other so that indirect calls do not miss any potential target.

5 Phase 2: Target Identification

This section presents how SMLTA identifies indirect call targets utilizing the type information stored in Type-Func Map, Type-Type Map and Type Lookup Maps.

5.1 Friend Type Discovery

For each indirect call, SMLTA examines whether the function pointer is a formal parameter to decide if fuzzy type should be
added to complete its multi-layer type. Once the multi-layer type of an indirect call is ascertained, we discover all of its friend types to ensure that the associated functions shared by the friend types will be identified as potential targets. Given that a multi-layer type can be partially transformed from another type, we extract all possible fragments, as defined in Definition 3, to support the exhaustive search of friend types relative to the entire multi-layer type of the indirect call.

**Definition 3** A fragment of multi-layer type T is one or multiple continuous layers in T.

A multi-layer type with N layers has \( \frac{(1+N) \times N}{2} \) possible fragments, as exemplified in Table 3. The Fragment column lists all possible fragments while before and after each fragment are respectively listed in Before-Part and After-Part columns. We can substitute each fragment with its friend types and concatenate these types with corresponding Before-Part and After-Part to generate friend types relative to the entire multi-layer type, thereby transforming the task of searching for friend types relative to the multi-layer type into discovering friend types for individual fragments.

For each fragment, the search of direct friend types is straightforward, which is achieved by querying Type-Type Map. However, the exhaustive search of indirect friend types can be challenging because these types may be buried in long, cyclic chains of information flow. To address this issue, SMLTA employs an exhaustive search algorithm adapted from Breath-First Search, which monitors the state of discovered friend types to bypass cycles in chains. This algorithm is detailed in Algorithm 1, where Frag represents a fragment and Fragft represents its friend types.

We prepare three sets, \( S_{\text{checked}} \), \( S_{\text{checking}} \), and \( S_{\text{unchecked}} \) to manage friend types in different states and prevent cyclic search. A friend type currently being used to query the Type-Type Map is held in \( S_{\text{checking}} \) and is moved to \( S_{\text{checked}} \) post-query. \( S_{\text{unchecked}} \) holds newly discovered friend types for the next round of search. Initially, the current Frag is placed into \( S_{\text{checking}} \). In each search iteration, we look for friend types for elements in \( S_{\text{checking}} \) by querying Type-Type Map and place newly discovered friend types to \( S_{\text{unchecked}} \). After each iteration, update the three sets for the following round of search. Repeat this process until \( S_{\text{checking}} \) is empty and \( S_{\text{checked}} \) is no more updated, indicating that all Fragft have been discovered.

<table>
<thead>
<tr>
<th>Before-Part</th>
<th>Fragment</th>
<th>After-Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 3: A table recording the fragments of multi-layer type A|B|C. A multi-layer type A|B|C with 3 layers has 6 possible fragments, including single-layer fragments, two-layer fragments, and three-layer fragments.

**Algorithm 1** Exhaustive search of Fragft

**Require:** Type-Type Map, Frag

**Ensure:** All Fragft are placed in \( S_{\text{checked}} \)

1: \( S_{\text{checked}} \leftarrow {} \)
2: \( S_{\text{checking}} \leftarrow \{\text{Frag}\} \)
3: \( S_{\text{unchecked}} \leftarrow {} \)
4: \( S_{\text{checked}}._{\text{origSize}} \leftarrow 0 \)
5: \( S_{\text{checked}}._{\text{newSize}} \leftarrow 0 \)
6: while \( S_{\text{checking}} \) is not empty do
7: for e in \( S_{\text{checking}} \) do
8: \( S_{\text{checked}} = \text{SetMerge}(S_{\text{checked}}, \text{Type-Type-Map[e]}) \)
9: \( S_{\text{checked}}._{\text{origSize}} = S_{\text{checked}}._{\text{size}}() \)
10: \( S_{\text{checked}} = \text{SetMerge}(S_{\text{checked}} \setminus S_{\text{checking}}) \)
11: \( S_{\text{checked}}._{\text{newSize}} = S_{\text{checked}}._{\text{size}}() \)
12: if \( S_{\text{checked}}._{\text{origSize}} == S_{\text{checked}}._{\text{newSize}} \) then return \( S_{\text{checked}} \)

Given friend types of each fragment, we generate the friend types relative to the entire multi-layer type of an indirect call by concatenation operations, and gather them in a set \( S \) along with the multi-layer type itself for type verification.

**5.2 Type Verification**

The elements in \( S \), are not exactly the multi-layer types that confine target functions of this indirect call due to three kinds of mismatches: 1) Some friend types generated through fragments do not exist among the recorded multi-layer types; 2) Some elements in \( S \), containing fuzzy type and index, apparently differ from but actually match with the recorded multi-layer types; 3) Some recorded multi-layer types, containing fuzzy type and index, apparently differ from but actually match with the elements in \( S \). Hence, we query Type Lookup Maps for type verification and collecting verified types, which is defined in Definition 4.

**Definition 4** A verified type is a multi-layer type that is recorded in Type-Func Map and Type Lookup Maps, and matches with an element in \( S \).

To perform type verification, we check each element in \( S \) to find verified types in Type Lookup Maps. Given a multi-layer type \( T \) in \( S \), first of all, we pass the first-layer type of \( T \) to The First Map to achieve a scope of second-layer types. Then, check whether the elements in this scope match with the second-layer type of \( T \). Concatenate the first-layer type with every matched second-layer type to generate a set of first-two-layer types. Lookup The Second Map to achieve a scope of third-layer types and find those who match with the third-layer type of \( T \). Following this working pattern to check the remaining layers until all verified types are retrieved from Type Lookup Maps. Finally, query Type-Func Map with verified types to find associated functions. The union of such functions are identified as potential targets of the indirect call.
6 Implementation

DEEPTYPE is built on LLVM 15.0 in 2.8k lines of C++ code. It supports C and C++ programs, providing particular advantages for programs that frequently employ composite data structures. This section delineates the enhancements in accuracy and performance from implementation perspective.

6.1 Special Handlings

In addition to the novel approach SMLTA, DEEPTYPE enhances accuracy from implementation aspect by addressing diverse instructions and rare code patterns. We identified four typical corners cases and applied special handlings to each, aiming to mitigate inaccuracy resulting from the imprecise processing of these corner cases.

6.1.1 Composite instructions

A composite instruction encapsulates one or more embedded instructions as operands. When analyzing composite instructions, DEEPTYPE cannot acquire complete type information due to the obscured types within the embedded instructions. Listing 5 demonstrates a composite instruction. It is a store instruction that encapsulates a bitcast as its second operand, which itself is also a composite instruction embedding a getelementptr instruction as its first operand.

To gather complete type information, we employ ordered trees to represent hierarchical structure of composite instructions and their embedded instructions, and conduct a systematic analysis of instructions within ordered trees from the lowest to the highest levels, ensuring that the types hidden in embedded instructions can be utilized to generate complete multi-layer types when analyzing their parent instructions. For example, we construct a three-level tree to represent the instructions in Listing 5. The store instruction occupies the root node in level 0, the bitcast instruction is situated in the second child node at level 1, and the getelementptr instruction resides in the first child node of the bitcast at level 2, as Figure 3 depicts. The getelementptr instruction returns a three-layer type int(*)* 1 struct.sqlite3_mutex_methods#0 1 struct.Sqlite3Config#13, which serves as the source type for the bitcast. So, we record it and the destination type int(*)* 1 struct.A as friend types each other in Type-Type Map. This relationship contributes to accuracy, but it could be missed without the special handling to composite instructions.

Listing 5: A composite instruction. The store instruction in sqlite is composite, incorporating an embedded bitcast instruction, which in turn is composite and contains an embedded getelementptr instruction.

Figure 3: Ordered tree of the instructions in Listing 5. The root node represents the composite instruction store. Its child nodes at level 1 denote the instruction’s operands. The second operand is another composite instruction bitcast. The child nodes at level 2 correspond to the operands of bitcast, among which the first one is a getelementptr instruction.

6.1.2 Anonymous structures

In LLVM bitcode, structs are sometimes anonymous when the specific names are not necessary or when they help to optimize the internal representation. This anonymity helps reduce IR size, but can lead to type mismatches, resulting in imprecise target identification.

We address this by assigning a unique identifier to each anonymous struct based on the sequence of member types. This identifier assists in linking anonymous structs to named equivalents elsewhere in the bitcode, thus mitigating inaccuracy caused by mismatches of anonymous structs. If no named equivalent is found, they are named as struct.anon, which conservatively matches with any struct, preventing missing targets resulted from anonymous structs.

6.1.3 Dead functions

The iterative updating and patching of applications often results in the presence of dead functions, which are declared but not invoked, thus will not be executed during runtime. The analysis of instructions within these dead functions generates redundant type information, leading to false positive targets and unnecessary performance overhead. To enhance both accuracy and efficiency, our analysis omits dead functions.

6.1.4 Empty Type

In LLVM IR, the notation "{}" is employed to denote an anonymous struct type when the specific details of the data structure are unnecessary and subsequently omitted. We name it as "empty type." Contrary to the anonymous structs discussed in Section 6.1.2, which merely lack names, the empty type is anonymous and contains no fields. However, what it represents can often be inferred from bitcast and other corresponding instructions in proximity. We categorize the code patterns involving empty types into two distinct classes, addressing each class with tailored solutions. Otherwise, the presence of empty type can impact the precision of our analysis as itself does not match with any struct.

As of October 2023, the latest version of TypeDive uses similar method to identify anonymous structs. However, it ignores those anonymous structs whose names never appear in the bitcode, which may lead to missing targets.
When the empty type serves as the destination type in a bitcast, we observed that any subsequent utilization of the empty type in IR corresponds to the usage of the source type in source code. Accordingly, we record the source type as a friend type relative to the empty type, mitigating missing targets that may result from type mismatch. Conversely, when the empty type is the source type in a bitcast, we observed that its outer layer types in IR are the outer layer types of the destination type in source code. Thus, we record the empty type’s outer layer types as those of the destination type’s, ensuring that complete multi-layer types are gathered for type matching and target identification.

6.2 Caches

To diminish the performance overhead, we deploy two caches aiming at reducing runtime cost without affecting accuracy. The first one is used to store the verified types of a multi-layer type so that the exhaustive search algorithm and type verification process only run once for each multi-layer type. The second one is used to store identified targets of an indirect call so that another indirect call with the same multi-layer type can be quickly resolved by accessing the cache.

7 Evaluation

We evaluate the effectiveness of DEEPTYPE in Section 7.1 and overhead in Section 7.2, comparing it with TypeDive (commit acb8f4c) since MLTA is the-state-of-the-art approach in type-based analysis. Additionally, we evaluate the contribution of SMLTA in accuracy in Section 7.3 and present its security impact through a case study in Section 7.4.

Our evaluation is conducted on Linux, 5 web servers and 14 user applications. The GNU Binutils-2.35 is a collection of binary tools. We selected 13 among 15 programs as the discarded ones barely use indirect calls.

SQLite-3.45.1 is a database engine which contains numerous multi-layer types. 5 server programs are nginx, httpd, openVPN, proftpd and sshd. We use Linux-5.1 as the benchmark to show the scalability of DEEPTYPE.

Experiments are conducted on Ubuntu 20.04 with 8-core Intel Core i9-9880H CPU @ 2.30GHz and 16GB DDR4 RAM. The benchmarks are compiled by WLLVM [54] with LLVM-15. We use -g -O0 flags to ensure that the generated bitcode contains debug information and type information, and that the instructions DEEPTYPE analyzes are not optimized out. Another flag -Xclang -no-opaque-pointers is used to disable opaque pointers so that pointers’ types are sustained.

To compare with TypeDive, we execute both tools on LLVM-15 and apply dead function elimination on them to make sure the benchmark bitcode analyzed by DEEPTYPE and TypeDive are exactly the same. In the experiments, we deploy 7-layer mappings in Type Lookup Maps, which are sufficient to archive the multi-layer types in our benchmarks. Appendix A shows how to determine the number of mappings.

7.1 Effectiveness of DEEPTYPE

The effectiveness of DEEPTYPE is demonstrated by its ability to narrow down the scope of indirect call targets. We use Average Number of Targets (ANT) as metric to quantitatively measure effectiveness, which is defined as:

\[
ANT = \frac{\text{Num}(T)}{\text{Num}(IC)},
\]

where \(\text{Num}(T)\) represents the total number of identified targets, \(\text{Num}(IC)\) represents the total number of indirect calls that have targets. While the metric in TypeDive paper [27] is also average number, it only takes into account the indirect calls whose multi-layer types have at least two layers. Their metric neglects the fact that single-layer types can also benefit from MLTA, and that some indirect calls have no target identified because they are not initialized or because DEEPTYPE and TypeDive miss targets due to inevitable obstacles in implementation, as elaborated in Section 8.1. Therefore, we define ANT to precisely evaluate the effectiveness. To clarify false positive (FP) and false negative (FN) subsequently used in this paper, we define them in Definition 5.

Definition 5 Given an indirect call, a false positive (FP) is a function erroneously included in the target set contrary to the ground truth. A false negative (FN) is a function erroneously excluded from the target set contrary to the ground truth.

Table 4 presents the ANT for the benchmarks tested by DEEPTYPE and TypeDive, and the reduction rate in ANT achieved by DEEPTYPE compared to TypeDive. Given that the binutils programs share numerous library functions, leading to analogous ANT value, we only list their average ANT. The details are available in Appendix B. The data in Table 4 indicates that DEEPTYPE reduces the ANT by 43.11% on average across most benchmarks, including binutils, httpd and linux. However, DEEPTYPE does not manage to decrease the ANT for nginx, openvpn and proftpd.

The reduction in ANT can be attributed to SMLTA and special handlings in DEEPTYPE. SMLTA follows the strong restriction that checks the entire multi-layer type of an indirect call to identify targets that match with it. In contrast, TypeDive employs MLTA which separately resolves each layer of a multi-layer type and calculates intersection to determine the target set, potentially leading to FPs. The special handlings in DEEPTYPE are tailored to address corner cases where type
information may be obscured, which enable DEEP_TYPE to extract more accurate type information, thereby reducing FPs.

To validate the reasons for ANT reduction, we conducted a manual analysis on objcopy, as it contains substantial yet manageable number of indirect calls. By manually examining the indirect calls for which DEEP_TYPE collects fewer targets than TypeDive, we confirmed the contributions of SMLTA and special handlings, and additionally unveiled another factor contributing to the reduction of ANT.

Field-sensitivity deployed in DEEP_TYPE enables the differentiation of distinct members within a composite data structure, even if they share the same type, which further refines indirect call targets. Despite the assertion of field-sensitivity in TypeDive paper, our manual analysis discovered FPs due to its field-insensitive. For instance, in a simplified scenario, a function is confined to $i64(i8*)* | struct.bfd_target#13$, whereas an indirect call has multi-layer type $i64(i8*)* | struct.bfd_target#23$, without any type propagation involved. TypeDive identifies this function as a target because its first and second layers respectively match $i64(i8*)* and struct.bfd_target$. By contrast, field-sensitive DEEP_TYPE mitigates such FPs. Thus, field-sensitivity is another reason for ANT reduction.

Table 4 also shows that DEEP_TYPE does not consistently reduce ANT, particularly in the cases of nginx and openvpn. There are two reasons for the increasing ANT. First, the adoption of fuzzy type leads to the conservative inclusion of all potentially matching types. It decreases FNs meanwhile inevitably brings in FPs, consequently raising the ANT value. Second, the special handlings enable DEEP_TYPE to extract more precise type information, diminishing both FPs and FNs. In situations where the reduction in FNs surpasses that in FPs within a program, the ANT value increases. Given that the rising ANT is attributed to the reduction in FNs, it is deduced that DEEP_TYPE is theoretically and practically effective in refining indirect call targets.

Figure 4 shows the distribution of indirect calls with different number of targets in Linux. We observe that DEEP_TYPE is capable of finding more valid targets for more indirect calls, thus DEEP_TYPE and DeepType-Cumu have more indirect calls than TypeDive and TypeDive-Cumu in most intervals. The main difference between DEEP_TYPE and TypeDive falls in intervals $[2,4)$ and $[4,8)$, the number of targets represented by these two intervals is much smaller than the average number of indirect call targets reported by TypeDive. As the number of indirect call targets increases, the difference between DEEP_TYPE and TypeDive gradually decreases and becomes trivial except for interval $[64,128)$, where DEEP_TYPE has more indirect calls, and interval $[128,256)$, where TypeDive has more indirect calls.

This trend indicates that the ratio of indirect calls with small target sets is lower in DEEP_TYPE. We define small by different values of threshold in Table 5. A target set is considered as small when its size is less than the threshold. We observe that when the threshold is 2 or 4, DEEP_TYPE has lower ratio of indirect calls with small target sets. However, this does not imply that TypeDive outperforms DEEP_TYPE for these indirect calls. At lower thresholds (i.e., 2 or 4), a single FP or FN can significantly impact whether an indirect call is categorized as with a small or large target set.

Additionally, we compiled the benchmarks on various optimization levels to assess the effectiveness of DEEP_TYPE across these levels. The data presented in Table 6 indicates that DEEP_TYPE reports higher ANT when the benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>DEEP_TYPE</th>
<th>TypeDive</th>
<th>Reduction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>binutils</td>
<td>2.47</td>
<td>10.98</td>
<td>77.50%</td>
</tr>
<tr>
<td>sqlite</td>
<td>6.24</td>
<td>8.32</td>
<td>25.00%</td>
</tr>
<tr>
<td>nginx</td>
<td>6.38</td>
<td>5.60</td>
<td>-13.93%</td>
</tr>
<tr>
<td>httpd</td>
<td>6.23</td>
<td>12.27</td>
<td>49.23%</td>
</tr>
<tr>
<td>openvpn</td>
<td>2.35</td>
<td>1.62</td>
<td>-45.06%</td>
</tr>
<tr>
<td>proftpd</td>
<td>3.10</td>
<td>2.96</td>
<td>-4.73%</td>
</tr>
<tr>
<td>sshd</td>
<td>5.43</td>
<td>5.57</td>
<td>2.51%</td>
</tr>
<tr>
<td>linux</td>
<td>9.74</td>
<td>25.17</td>
<td>61.30%</td>
</tr>
</tbody>
</table>

Table 4: Average number of indirect call targets. This table shows the average number of indirect call targets, and the reduction rate produced by DEEP_TYPE over TypeDive. binutils shows the average of the 13 programs in binutils, the detailed data of each is elaborated in Appendix B.
The definition of small depends on the threshold. If the indirect call’s target set size is smaller than the threshold value, this indirect call is considered as with small target sets. The last two columns respectively show the ratios of indirect calls with small target sets in DEEPType and TypeDive.

Table 6: The effectiveness of DEEPType on different optimization levels. This table shows the ANT reported by DEEPType when analyzing benchmarks respectively compiled with optimization level O0, O1, O2 and O3.

<table>
<thead>
<tr>
<th>Program</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>binutils</td>
<td>2.47</td>
<td>3.20</td>
<td>3.20</td>
<td>3.13</td>
</tr>
<tr>
<td>sqlite</td>
<td>6.24</td>
<td>6.46</td>
<td>6.48</td>
<td>6.56</td>
</tr>
<tr>
<td>nginx</td>
<td>6.38</td>
<td>8.00</td>
<td>8.02</td>
<td>7.99</td>
</tr>
<tr>
<td>httpd</td>
<td>6.23</td>
<td>6.23</td>
<td>6.23</td>
<td>6.23</td>
</tr>
<tr>
<td>openvpn</td>
<td>2.35</td>
<td>2.80</td>
<td>2.45</td>
<td>2.45</td>
</tr>
<tr>
<td>proftpd</td>
<td>3.10</td>
<td>3.10</td>
<td>3.10</td>
<td>3.10</td>
</tr>
<tr>
<td>sshd</td>
<td>5.43</td>
<td>5.43</td>
<td>5.43</td>
<td>5.43</td>
</tr>
<tr>
<td>linux</td>
<td>9.74</td>
<td>9.74</td>
<td>9.74</td>
<td>9.74</td>
</tr>
</tbody>
</table>

To deduce the primary source of runtime overhead and reveal the reason for DEEPType’s efficiency, we separately measured the execution time of information collection and target identification phases, considering the shared general workflow of all tools. The experiments were conducted on linux benchmark, which manifests noticeable differences among three tools. According to Figure 6, the percentages of runtime overhead incurred during target identification exhibit a progressive increase among DEEPType, DT-nocache, and TyepDive, standing at 20.6%, 44.2%, and 65.0%, respectively. Thus, the lower overhead of DEEPType should owe to the target identification phase, wherein DEEPType straightforwardly deals with the entire multi-layer type of each indirect call. In contrast, TypeDive addresses each two-layer type within the multi-layer type and calculates intersections, incurring additional runtime overhead. The fact that DEEPType refines indirect call targets also indicates that TypeDive consumes extra computational resources on FP.

An additional observation reveals that the distinction in runtime overhead is evident in binutils, sqlite and linux, whereas it is negligible in the five server programs. The reason for this distinction is the higher prevalence of multi-layer types with more layers in binutils, sqlite and linux as illustrated in Figure 7. The efficiency advantage of DEEPType over TypeDive is primarily attributed to DEEPType’s one-time resolution on the entire multi-layer type, in contrast to the laborious resolution in TypeDive for each two-layer type. It is noteworthy that a multi-layer type may consist of multiple two-layer types. Consequently, the benchmarks containing a greater number of multi-layer types with more layers amplify the runtime overhead discrepancy between DEEPType and TypeDive.

Typically, memory overhead is not a major concern in static analysis, as tools that offer efficient performance without compromising precision are often favored. We still measure the memory overhead of DEEPType using Massif tool in Valgrind [31] tool suite with -p a s -h eap=yes option enabled, to measure all the memory used, and compare it with TypeDive for a thorough performance assessment.

Figure 8 shows the memory overhead of DEEPType and TypeDive. Regarding user applications and server programs in the benchmarks, both tools exhibit memory overhead below 150 MB. However, in the context of the Linux kernel, both DEEPType and TypeDive demonstrate higher memory overheads ranging from 4.2 GB to 4.3 GB. This discrepancy is attributed to the larger size of kernel program, which involves
more multi-layer types. As a result, both tools record a greater volume of type information, leading to increased sizes of data structures and consequently consuming more memory spaces.

**DeepType** shows lower memory overhead than **TypeDive**, while the subtle difference between two tools remains consistent across all benchmarks. This consistency is attributed to the comparable memory space occupied by the data structures in two tools. Although **DeepType** records entire multi-layer types, which have larger sizes than two-layer types, it allocates fewer entries in maps for storage. The difference is due to the additional memory space utilized in **TypeDive** for recording escaping types, which is designed to overcome the third challenge, as elaborated in Section 2. In contrast, **DeepType** adopts the exhaustive search algorithm which does not consume so much memory space as escaping types.

### 7.3 Contribution of SMLTA in Accuracy

As described in Section 7.1, **DeepType** is capable to narrow down the scope of indirect call targets. This capability owes to SMLTA and special handlings to corner cases. To further investigate the impact of SMLTA on effectiveness, we disabled the special handlings in **DeepType**, resulting in a variant denoted as DT-noSH. Different ANT values of **DeepType** and DT-noSH exhibit the impact of special handlings, thus revealing the contribution of SMLTA.

Table 7 presents the ANT reported by **DeepType** and DT-noSH. The ANT of DT-noSH is close to that of **DeepType**, indicating that disabling the special handlings has minimal impact on the effectiveness of **DeepType**, which reveals the primary role of SMLTA in accuracy improvement. The rationale behind the statistics is that the special handlings are specifically implemented to address corner cases. Without them, **DeepType** generates slightly more FPs and FNs. Given the definition of ANT, minor fluctuations in the total number of indirect call targets do not significantly alter ANT value when the volume of indirect calls is relatively high. Nonetheless, these special handlings remain crucial, as they play a vital role in mitigating the FPs and FNs that are orthogonal to SMLTA.

Recall that there are four special handlings. First, the systematic analysis of composite instructions facilitates the generation of complete multi-layer types. Among all the benchmarks, nginx and linux exhibit the most significant improvement owing to this special handling, given their relatively higher prevalence of composite instructions. Second, linking anonymous structs with named equivalents enables **DeepType** to accurately recognize and match multi-layer types. We observe that openvpn and linux obtain the most ben-
Table 7: ANT values of DEEPTYPE, DT-noSH and DT-weak. DT-noSH exhibits the contribution of SMLTA. DT-weak shows the impact of storing entire multi-layer types in Type-Func Map.

<table>
<thead>
<tr>
<th>Program</th>
<th>DEEPTYPE</th>
<th>DT-noSH</th>
<th>DT-weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>binutils</td>
<td>2.47</td>
<td>2.48</td>
<td>2.70</td>
</tr>
<tr>
<td>sqlite</td>
<td>6.24</td>
<td>6.33</td>
<td>6.97</td>
</tr>
<tr>
<td>nginx</td>
<td>6.38</td>
<td>8.62</td>
<td>12.99</td>
</tr>
<tr>
<td>httpd</td>
<td>6.23</td>
<td>6.23</td>
<td>7.66</td>
</tr>
<tr>
<td>openvpn</td>
<td>2.35</td>
<td>2.39</td>
<td>2.35</td>
</tr>
<tr>
<td>proftpd</td>
<td>3.10</td>
<td>3.13</td>
<td>4.22</td>
</tr>
<tr>
<td>sshd</td>
<td>5.43</td>
<td>5.42</td>
<td>5.43</td>
</tr>
<tr>
<td>linux</td>
<td>9.74</td>
<td>9.72</td>
<td>13.09</td>
</tr>
</tbody>
</table>

Listing 6: CVE-2023-43642 vulnerable code. Line 3 is vulnerable to out-of-bounds access. An exploit for this vulnerability can overwrite a function pointer in glib to gain code execution.

```c
void track_set_index (...) {
    ...
    track->index[i] = ind;
}

static gboolean get_file_metadata (...) {
    TrackerExtractInfo *info;
    ...
    tracker_extract_info_unref(info);
    ...
}

gboolean g_option_context_parse (...) {
    ...
    if (!(* group->pre_parse_func) (context, group,  
        group->user_data, error))
    ...
}
```

### 7.4 Case Study

While SMLTA is a fundamental tool applicable across various security-related fields (e.g., static bug detection, symbolic execution, fuzzing, and etc.), we demonstrate its security impact through the example of CFI enforcement.

CVE-2023-43641 [11] is an out-of-bounds access that enables arbitrary write in libcue. As detailed in Listing 6 at line 3, both the value of index `i` and `ind` can be controlled by attacker. By setting the index `i` negative, an attacker can achieve arbitrary write through preparing the value of `ind`. The exploit [2] utilized this vulnerability to corrupt a function pointer in glib (cross-referenced by libcue) to achieve arbitrary code execution. To be specific, the exploit overwrites the heap object `info` (line 7), which is allocated in a function invoked by `track_set_index` (line 1). This overwrite enables the corruption of function pointer `pre_parse_func` (line 15) in glib, and the corrupted function pointer is subsequently used to call the `initable_init` function, which behaves similarly to the `system` function, enabling arbitrary code execution.

Similar to many typical CVE exploits, this exploit assumes Control Flow Integrity (CFI) is not deployed. The function pointer `pre_parse_func` has a type mismatch with the target function `initable_init`, making the exploit preventable by both MLTA and SMLTA. However, this does not imply that the vulnerability can be completely mitigated by either MLTA or SMLTA, as attackers may still conduct exploits by corrupting alternative function pointers.

We further examined glib and revealed 5 function pointers that can bypass MLTA, see Table 8. MLTA fails to prevent the exploits that corrupting these function pointers because it identifies the function `initable_init` as a valid target for these function pointers, though it is a FP in fact. This FP can be attributed to MLTA’s approach of splitting multi-layer types, which confines the function `initable_init` respectively to `gboolean (GInitable*, GCancellable*, GError**)`.*
which produces FNs. First, fuzzy type and fuzzy index stand which has already been proved [30, 33, 37, 53, 59]. In general, while SMLTA can prevent these exploits. To differentiate two function pointers named “callback” in separate functions, one is denoted as “callback*”.

and struct._GInitableIface with index 1, weakening the restriction of multi-layer type matching. For example, traverse_func in Table 8 has type gboolean (Ginitable*, GCancellable*, GError**)*, which matches with gboolean (GInitable*, GCancellableable*, GError**)* due to information flow in glib. Thus, MLTA identifies initable_init as a potential target, enabling attackers to rewrite the function pointer with the address of initable_init and achieving arbitrary code execution.

In contrary, SMLTA is able to prevent these exploits because it strictly confines initable_init to its entire multi-layer type gboolean (GInitable*, GCancellableable*, GError**)* l struct._GInitableIface#1, which does not match with any function pointer in Table 8. Consequently, SMLTA effectively reduces the attack surface of control-flow hijacking attacks utilizing this vulnerability. By limiting the attacker’s ability to corrupt function pointers, SMLTA offers a higher level of security compared to MLTA in CFI implementation.

<table>
<thead>
<tr>
<th>Function Pointers</th>
<th>MLTA</th>
<th>SMLTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>callback</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>callback*</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>traverse_func</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>func</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>predicate</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 8: The capability of MLTA and SMLTA in preventing exploits. The listed function pointers, located in glib, can be corrupted through the vulnerability. MLTA fails to prevent the exploits through the 5 function pointers while SMLTA can prevent these exploits. To differentiate two function pointers named “callback” in separate functions, one is denoted as “callback*”.

8.2 Limitations

The implementation of DEEP-TYPE does not robustly address all corner cases in real-world programs. Besides the ones addressed in Section 6, some corner cases are out of our scope. For example, LLVM IR only contains type information but omits instantiated values of the members in global variables with composite type. If a function pointer as a member of a global variable is initialized at the global scope, the function assigned to it is apparent in source code but does not appear in the corresponding IR, thus does not appear in bitcode.9 Given this fact, DEEP-TYPE is unable to record the mappings between multi-layer types and functions for such initialization instructions, resulting in missing targets for involved indirect calls. Take binutils programs as another example, some indirect call targets are functions in the GNU linker, LD, while we use Clang to compile the programs which uses LLD as linker.

Albeit the accuracy improvement contributed by SMLTA and special handlings, DEEP-TYPE still exhibits deficiencies in terms of accuracy. For small programs that contain few complicated multi-layer types, DEEP-TYPE is limited in reducing FPs compared to TypeDive meanwhile possibly yielding extra FPs instead due to the conservative design and implementation choices for soundness purpose. In addition, if several function pointers have the same multi-layer type but points to different functions, SMLTA produces FPs in this scenario.

In evaluation aspect, there is no standard scale available to calculate the statistics such as FP rate, FN rate and accuracy, due to the absence of ground truth. Although “pseudo ground truth” [24] is a feasible solution, it highly depends on the precision of dynamic analysis adopted. If the dynamic analysis result is not proved to be extremely close to the ground truth, the numerical data of soundness and precision relative to the pseudo ground truth is not convincing subsequently. We use ANT as metric to reflect the accuracy improvement. But it is convincing only in the context that both SMLTA and MLTA pertain to type-based analysis and the novel designs in DEEP-TYPE does not produce FNs.

8.3 Future Work

SMLTA generates FPs when multiple functions share the same multi-layer type with an indirect call. This is a limitation intrinsic to type-based analysis that relies solely on type verification. To enhance accuracy, we intend to integrate data-flow analysis with SMLTA, assessing both value and multi-layer type to further constrain potential targets, thereby reducing the FPs inherent to SMLTA. To maintain a balance between accuracy and performance, this data-flow analysis will be designed to be lightweight, focusing on intra-procedural analysis.

9Bitcode is a binary encoding of LLVM IR.
Motivated by the absence of a standard and reliable metric for comparing various approaches, we also plan to develop a benchmark that includes a comprehensive ground truth. This ground truth will encompass, but is not limited to, indirect call targets, alias pointers, and value sets. This will enable a wide range of basic tools used in both static and dynamic analysis to evaluate their effectiveness, identify the fundamental reasons for any inaccuracies, and enhance their design and implementation accordingly.

9 Related Work

Conservative solutions. Zhang et al. propose a protection mechanism called CCFIR [65] which implements a policy pertaining to indirect control transfers. It prohibits any indirect calls/jumps to locations other than those included in a white-list, which is derived from the relocation tables of ASLR [55] that provides absolute addresses of valid functions. Similarly, Zhang and Sekar [67] conservatively count all valid functions as targets given the assumption that source code is not available. These solutions are exceedingly conservative and may produce numerous false positives targets. Type-based analysis. In cases where the source code of a program is available, types of function pointers are utilized to deduce potential targets. Each indirect call is restricted to being directed towards only the functions with the same type. Tice et al. present VTV and IFCC that provide forward-edge CFI protection respectively for GCC and LLVM [49]. VTV conducts a validation process to ensure the correctness of the vtable pointer used for a virtual call by confirming the vtable pointer to be used for the indirect call points either to the vtable for the static type of the object, or to a vtable for one of its descendant classes. IFCC forces a function pointer into the right function-pointer set in jump tables to mitigate CFI violations and each function-pointer set has a specific function type signature. Niu and Tan present MCFI [34] where an indirect call is allowed to call any address-taken function whose type is structurally equivalent to the function pointer’s type. Victor et al. propose TypeArmor [52] to reduce the number of targets for indirect calls where function type and argument number is checked to identify valid targets. MLTA [27] utilizes multi-layer type, which includes function type signature and type of composite data structures, to significantly refine indirect call targets. The majority of the prior work on resolving C++ virtual functions depend on class hierarchy analysis combined with object type information [15, 21, 40, 49, 64]. These approaches employ an expanded single-layer type to identify targets, but our approach can further improve the accuracy by utilizing types of multiple involved data structures. Data tracking Analysis. Ge et al. utilize taint analysis in their fine-grained CFI to identify targets for kernel software [14]. They begin with a function and taint all function pointers that are initialized with the function or the tainted function pointers. If a function pointer is in a structure, they taint the field for all memory objects of that structure’s type. This policy enables fine-grained CFI to leverage the information of directly related data structure instead of only considering the function pointer itself, but it is still less precise than SMLTA which uses type information of all involved data structures to restrict targets. Kim et al. use block-based pointer analysis (BPA) [23] to identify indirect call targets, which generates memory blocks for heaps, stack frames, and global data sections following specific rules and perform pointer analysis in each memory block to infer points-to-relationships. They also upgrade BPA by offset-sensitivity in BinPointer [24], which tracks the offset in each memory block for more fine-grained pointer analysis. Compared with strong MLTA, such approach is deficient in handling type casting, which will cause missing targets. Another limitation is that it does not support C++. CF-GAccurate [46] interleave four techniques including forced execution, backward slicing, symbolic execution and value set analysis to construct CFG, which is precise at the expense of high performance overhead.

Dynamic Analysis. Many other solutions [12, 13, 16, 35, 51] use dynamic analysis which relies on runtime information to identify indirect call targets. These solutions address a distinct problem area, which is the assurance of the validity of a runtime target. Conversely, our work statically deduces the range of potential targets for each indirect call.

10 Conclusion

In this paper, we have introduced strong multi-layer type analysis (SMLTA), a novel approach in refining indirect call targets, that thoroughly utilizes type information provided by multi-layer types. It treats the entire multi-layer type as a basic unit for information storage and type verification to improve accuracy. SMLTA resolves relationships between multi-layer types, exhaustively discovers friend types for indirect calls, and employs fuzzy type to overcome the challenges in indirect call target identification using multi-layer types. Additionally, multi-layer mappings are deployed to hierarchically archive multi-layer types for quick access. We implemented SMLTA in DEEPTYPE, which is equipped with special handlings to address diverse code patterns and corner cases. DEEPTYPE is scalable to large applications with superior effectiveness as well as performance. The experiment results showed that DEEPTYPE narrows down the scope of indirect call targets by 43.11% on average across most benchmarks, reduces runtime overhead by 37.02% on average and consumes less memory compared to TypeDive. A case study in CVE exploit demonstrated that SMLTA is more powerful than MLTA in reducing attack surface and preventing exploits. However, the intrinsic limitation of type-based analysis can still produce false positive targets. We leave it as future work to further improve accuracy through lightweight data-flow analysis.
Acknowledgment

We thank our shepherd and the anonymous reviewers for their insightful comments. Thanks to Taegyu Kim and Kaiming Huang for their suggestions and help. This work is supported in part by National Science Foundation (NSF) under grants CNS-2247652 and CNS-1652790.

References


The number should neither be too large to guarantee the performance of the multi-layer mappings to hierarchically archive them as described in Section 4.2. The question is how many mappings are sufficient in Type Lookup Maps? To efficiently organize and access the collected multi-layer types, we use multi-layer mappings to hierarchically archive them as described in Section 4.2. The number should neither be too large to guarantee the performance of Type Lookup Maps nor too small to hold the multi-layer types. So, we decide the number of mappings conforming to the multi-layer types in benchmarks.

Table 9 shows the number of multi-layer types with different layer counts. Across all benchmarks, the multi-layer types in them contain at most 8 layers, which means 7-layer mappings are sufficient to record these multi-layer types. Thus, we adopt 7-layer mappings in our implementation. If multi-layer types with more than 8 layers are common in other programs, more mappings can be deployed in DEEPTYPE to supply extra precision in practice.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Total</th>
<th>&lt;=8 layers</th>
<th>&gt; 8 Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>binutils</td>
<td>311*</td>
<td>311*</td>
<td>0</td>
</tr>
<tr>
<td>sqlite</td>
<td>220</td>
<td>220</td>
<td>0</td>
</tr>
<tr>
<td>nginx</td>
<td>101</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>httpd</td>
<td>88</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>openssl</td>
<td>43</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>proftpd</td>
<td>88</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>sshd</td>
<td>46</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>linux</td>
<td>5,447</td>
<td>5,447</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9: Number of multi-layer types with different layer counts. 311* indicates the average of binutils programs.

B Binutils Test results

Binutils programs consist of similar indirect calls as they utilize shared library function, thus displaying similar results. For someone who is interested in, we provide detailed data for each individual program in Table 10.

<table>
<thead>
<tr>
<th>Program</th>
<th>DEEPTYPE</th>
<th>DT-weak</th>
<th>TypeDive</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr2line</td>
<td>2.38</td>
<td>2.62</td>
<td>8.60</td>
</tr>
<tr>
<td>ar</td>
<td>2.45</td>
<td>2.69</td>
<td>12.73</td>
</tr>
<tr>
<td>bfdtest1</td>
<td>2.38</td>
<td>2.62</td>
<td>12.89</td>
</tr>
<tr>
<td>bfdtest2</td>
<td>2.38</td>
<td>2.63</td>
<td>12.89</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>2.37</td>
<td>2.62</td>
<td>8.64</td>
</tr>
<tr>
<td>nm-new</td>
<td>2.48</td>
<td>2.72</td>
<td>13.01</td>
</tr>
<tr>
<td>objcopy</td>
<td>2.63</td>
<td>2.87</td>
<td>13.08</td>
</tr>
<tr>
<td>objdump</td>
<td>2.95</td>
<td>3.19</td>
<td>11.26</td>
</tr>
<tr>
<td>ranlib</td>
<td>2.45</td>
<td>2.69</td>
<td>12.73</td>
</tr>
<tr>
<td>readelf</td>
<td>2.29</td>
<td>2.29</td>
<td>2.30</td>
</tr>
<tr>
<td>size</td>
<td>2.39</td>
<td>2.64</td>
<td>12.95</td>
</tr>
<tr>
<td>strings</td>
<td>2.37</td>
<td>2.62</td>
<td>8.64</td>
</tr>
<tr>
<td>strip-new</td>
<td>2.63</td>
<td>2.87</td>
<td>13.08</td>
</tr>
</tbody>
</table>

Table 10: Average number of targets for each program in binutils.

Appendix

A Multi-Layer Mappings

To efficiently organize and access the collected multi-layer types, we use multi-layer mappings to hierarchically archive them as described in Section 4.2. The question is how many mappings are sufficient in Type Lookup Maps? The number should neither be too large to guarantee the performance of DEEPTYPE nor too small to hold the multi-layer types. So, we decide the number of mappings conforming to the multi-layer types in benchmarks.