Type-Alias Analysis: Enabling LLVM IR with Accurate Types

JINMENG ZHOU* and ZIYUE PAN*, Zhejiang University, China
WENBO SHEN†, Zhejiang University, China
XINGKAI WANG, Zhejiang University, China
KANGJIE LU, University of Minnesota Twin Cities, USA
ZHIYUN QIAN, University of California, Riverside, USA

LLVM Intermediate Representation (IR) underpins the LLVM compiler infrastructure, offering a strong type system and a static single-assignment (SSA) form that are well-suited for program analysis. However, its single-type design assigns exactly one type to each IR variable, even when the variable may legitimately correspond to multiple types. The recent introduction of opaque pointers exacerbates this limitation: all pointers in the IR are uniformly represented with a generic pointer type (`ptr`) that erases concrete pointee type information, making many type-based analyses ineffective.

To address the limitations of single-type design, we introduce type-alias analysis, a multiple-type design that maintains type-alias sets for IR variables and infers types across IR instructions. We have developed TypeCopilot, a prototype that recovers concrete pointee types for opaque-pointer-enabled LLVM IR generated from C programs. TypeCopilot achieves 98.57% accuracy with 94.98% coverage, allowing existing analysis tools to retain their effectiveness despite the adoption of opaque pointers. To foster further research and security applications, we have open-sourced TypeCopilot, providing the community with a practical foundation for precise, type-aware security analyses on modern LLVM IR.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Additional Key Words and Phrases: LLVM IR, Type System, Opaque Pointer

ACM Reference Format:
Jinmeng Zhou, Ziyue Pan, Wenbo Shen, Xingkai Wang, Kangjie Lu, and Zhiyun Qian. 2025. Type-Alias Analysis: Enabling LLVM IR with Accurate Types. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA097 (July 2025), 24 pages. https://doi.org/10.1145/3728974

1 Introduction

LLVM Intermediate Representation (IR) lies at the heart of the LLVM ecosystem, and its Static Single Assignment (SSA) form enables rigorous analysis on data and control flows. Consequently, LLVM IR has become the foundation of numerous static-analysis frameworks. In this setting, the IR's precise type information—particularly its pointer types—serves as an indispensable pillar for sophisticated security analyses, including bug detection and exploitation [12, 24, 33, 48, 56], control-flow integrity [5, 10, 23, 31, 52, 53, 57], isolation [30, 32, 35], and defenses to protect critical

*Both authors contributed equally to this research.
†Wenbo Shen is the corresponding author.

Authors' Contact Information: Jinmeng Zhou, 11921110@zju.edu.cn; Ziyue Pan, Zhejiang University, Hangzhou, China, ziyuepan@zju.edu.cn; Wenbo Shen, Zhejiang University, Hangzhou, China, shenwenbo@zju.edu.cn; Xingkai Wang, Zhejiang University, Hangzhou, China, bittervan@zju.edu.cn; Kangjie Lu, University of Minnesota Twin Cities, Minneapolis, USA, kjlu@umn.edu; Zhiyun Qian, University of California, Riverside, Riverside, USA, zhiyunq@cs.ucr.edu.

ACM 2994-970X/2025/7-ARTISSTA097
https://doi.org/10.1145/3728974

data [21, 25, 36, 42, 52]. The effectiveness of these security analyses depends on the assumption that *pointer types in the IR correctly correspond to the underlying memory types (pointee types).*

However, this assumption may not always hold true due to the single-type limitation. LLVM IR supports defining each variable with a single type, and an IR variable typically corresponds to a single type. However, a pointer can have multiple pointee types in some cases. For example, multiple variables with different types can be assigned to the same pointer via different execution paths, with the type only being determined at runtime. Static analysis should consider these types from different paths as potential candidates.

In earlier versions of LLVM IR, the pointers are typed with concrete pointee types. The existing single-type design disallows annotating an IR variable with multiple types. Not all the concrete pointee types can be obtained for a pointer with multiple types. As a result, when static analysis attempts to identify objects of certain types by matching pointer types, it may overlook certain cases due to such challenges.

Even worse, all IR pointers discard pointee types and become opaque [38] in LLVM version 15 and later. As a pointer may correspond to multiple types, developers must expend significant effort maintaining these types and casting them back and forth. Ultimately, the LLVM community decided to eliminate pointee types of pointers. To support the single-type design, all typed pointers are replaced by generic pointer types, known as opaque pointers (`ptr`). As illustrated in Figure 1, `cred *` pointer is compiled as `ptr` in struct type definitions and function parameters. *The use of opaque pointers does not affect compiler functionality, but it can significantly undermine the effectiveness of security analysis.* Opaque pointers can point to any object, making pointer-type-based analysis schemes ineffective with massive false positives. Given the importance of LLVM IR types, it is essential to address the problems with a single-type design and provide more reliable types for security analyses. To our knowledge, no such solution currently exists.

To provide accurate and concrete types for security analyses, we introduce a new type-alias analysis featuring a multiple-type design. This approach collects multiple possible types into a type-alias set for an IR variable, representing its actual underlying memory types. We develop a new technique called *Alias-Rule-based Type Inference (short as TypeInfer)* to infer accurate types for IR variables among the complex propagation in IR instructions.

Leveraging this technique, we design a new type inference framework named TypeCopilot to provide accurate types for static analyses. TypeCopilot is applicable to LLVM IR with opaque pointers that are compiled from C language source code. Since all pointee types are lost in LLVM IR with opaque pointers, TypeCopilot seeks other reliable types as sources. Specifically, it restores accurate type sources from a high-level language for certain IR variables and then applies *TypeInfer*, using these restored variables as starting points to infer more types. Moreover, we implement TypeCopilot as LLVM passes to integrate it into other security analysis tools easily.

To thoroughly assess the effectiveness of TypeCopilot, we implement it on LLVM-16 with opaque pointers enabled. We evaluate the accuracy and coverage of TypeCopilot using three popular user libraries and the Linux kernel. Our results show that TypeCopilot achieves an overall accuracy of 94.23% under TBAA ground truth or 98.57% under CodeQL ground truth and a coverage of 94.98%. Compared to the baseline with opaque pointers, TypeCopilot improves accuracy by 79.62 percentage points and 52.66 percentage points under two ground truths, respectively. Besides, it improves coverage by 20.60 percentage points on average. Furthermore, we integrate TypeCopilot with existing tools, enabling them to maintain their effectiveness on IR with opaque pointers. In summary, we make the following contributions:

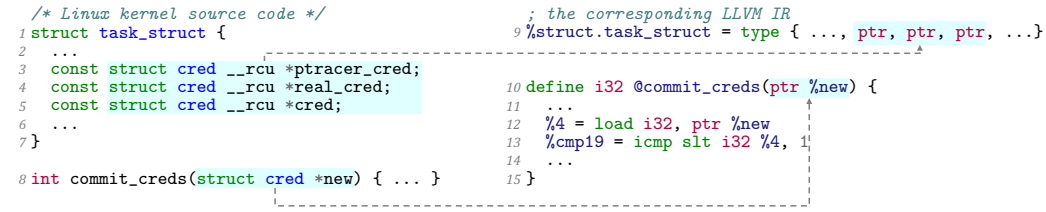


Fig. 1. A code snippet in the Linux kernel is compiled to the opaque-pointer IR.

- **New technique with multiple-type design.** We formalize type propagation in LLVM IR as a set of *type-alias rules*. On this basis, we further design *TypeInfer*—a new technique that infers precise type-alias sets for every IR variable.
- **New system.** We develop a type inference framework, named TypeCopilot, to support security analyses. By implementing *TypeInfer* and a new method of type restoration, TypeCopilot can infer types on the opaque-pointer IR. Moreover, its modular design allows seamless integration with existing security-analysis pipelines.
- **Practical evaluation.** Tested on widely deployed user-space libraries and the Linux kernel, TypeCopilot achieves 98.57% accuracy and 94.98% coverage, significantly improving the effectiveness of downstream security tools. To benefit the community, we open-source TypeCopilot and the dataset at https://github.com/ZJU-SEC/TypeCopilot.

2 Background

LLVM is a widely used compiler framework that integrates not only basic compilation but also analysis passes, thanks to the LLVM intermediate representation (LLVM IR) [19]. The LLVM IR facilitates many program analysis research due to its well-defined Static Single Assignment (SSA) format, modular design, and clean representation of high-level language properties [26].

Single-Type Design of LLVM IR's Type System. LLVM IR features a strong type system, where each IR variable is declared with a single explicit type [29]. It defines a single-value type consisting of primitive types and pointers, similar to the general understanding of scalar types. Aggregate types contain one or more fields of single-value types, such as structures and arrays. LLVM IR benefits static analysis since most types in the C language are consistent with those in LLVM IR. And LLVM exposes easy-to-use APIs (e.g., `getType()`) for users to retrieve the type information.

LLVM's strong type system disallows implicit type conversions, and all variables are defined with a single explicit type. For example, a variable `int *b` can be implicitly converted to another type in a conditional branch (e.g., `if (a == b)`) in the C language, where `a` is a generic pointer (`void *`). However, an explicit cast is required in LLVM IR, involving a derived variable (`%tmp = bitcast int* %b to void*`). It then compares the two variables to determine if they are equal (`%result = ICMP eq void* %a, %tmp`). To clarify, we define ***declared IR variables*** as the ones that directly correspond to the objects explicitly defined in the source code, such as `%a` and `%b`. On the contrary, ***derived IR variables*** do not directly correspond to source-code objects.

Opaque Pointers. In LLVM 14 and earlier, pointers are ***typed***, which consist of concrete base types and pointer indirection levels. For example, the base type of `i32 *` is `i32`, and the pointer indirection level is one. Since LLVM 15, the LLVM community has introduced ***opaque pointers*** [38] as a default feature. All pointers share the same type, i.e., `ptr` (a special kind of generic pointer).

```
/* Linux kernel source code */
1  struct io_ring_ctx {
2    ...
3    struct io_rings      *rings;
4    struct io_uring_sqe *sq_sqes;
5    ...
6  };

7  int io_uring_mmap(struct file *file,
     ↪  struct vm_area_struct *vma) {

8    ...
9    void *ptr;
10   struct io_ring_ctx *ctx =
       ↪  file->private_data;
11   ...

12   switch (offset) {
13     case IORING_OFF_SQ_RING:
14     case IORING_OFF_CQ_RING:
15       ptr = ctx->rings;
16       ...
17     case IORING_OFF_SQES:
18       ptr = ctx->sq_sqes;
19       ...
20   ...
21 }
```

```
; the corresponding LLVM IR
22 %struct.io_ring_ctx = type {
23   ...
24   %struct.io_rings*,
25   %struct.io_uring_sqe*,
26   ...
27 }

28 define i32 @io_uring_mmap(%struct.file* %file,
     ↪  %struct.vm_area_struct* %vma) {
29   ...
30   %4 = load %struct.io_ring_ctx*, %struct.io_ring_ctx** %3
31   ...
32 sw.bb.i:
33   %rings.i = gep %struct.io_ring_ctx, %struct.io_ring_ctx* %4,
       ↪  i64 0, i32 0, i32 1
34   %5 = bitcast %struct.io_rings** %rings.i to i8**
35   ...
36 sw.bb1.i:
37   %sq_sqes.i = gep %struct.io_ring_ctx, %struct.io_ring_ctx* %4,
       ↪  i64 0, i32 1, i32 2
38   %6 = bitcast %struct.io_uring_sqe** %sq_sqes.i to i8**
39   ...
40 sw.epilog.i:
41   %ptr.i.0.in = phi i8** [ %6, %sw.bb1.i ], [ %5, %sw.bb.i]
42   ...
43 }
```

Fig. 2. An example shows an IR variable has multiple types, which is not supported in the single-type design.

Despite the change, opaque pointers remain compatible with LLVM's single-type system, and LLVM APIs (e.g., `getType()`) still return types. Yet such opaque pointers also make important structures' types invisible to static analyses. For example, Figure 1 shows a security-critical structure type `task_struct` for process management, which has three credential fields with type `cred *` at line *3-5*. However, the type of all these fields in IR becomes `ptr` at line *9*, losing the original semantics. In addition to types in structure definitions, IR variables defined with pointer types also become `ptr`, such as the first parameter in `commit_creds` of type `cred *` at line *10*.

3 Motivation

LLVM IR types are widely used in program analyses [5, 25, 30–32, 35, 48, 57]. For example, to protect credentials in the Linux kernel, security researchers often identify IR variables of the type `struct cred *` and trace their propagation [42, 50]. This makes the accuracy of LLVM IR types crucial for the effectiveness of these analyses. In other words, these studies rely on the assumption that *pointer types in LLVM IR accurately correspond to the types of underlying pointed memory.*

However, our findings suggest that this assumption may not always hold true, presenting challenges for static analysis. This is mainly because the single-type design has prevented it from precisely representing all underlying memory types. This single-type limitation differs in typed-pointer IR and opaque-pointer IR.

3.1 Limitation of the Single-Type Design

Restrictive typed pointers. LLVM 14 and earlier support that a pointer has a single concrete pointee type. This design works for most IR variables. However, variables may have multiple types in some cases, and a single-type design naturally cannot represent multiple ones. To be compatible with the single-type design, there are two kinds of type representation methods.

First, if a variable corresponds to multiple types in certain cases, it may be forced to be a general type to support a single-type design, losing the concrete pointee information. As shown in Figure 2, the variable `ptr` of type `void *` is assigned two different types (`struct io_rings *` and `struct io_uring_sqe *`) in separate conditional branches (line *15* and line *18*). This is usually translated into a phi node in LLVM IR, as illustrated at line *41*. As a variable can only have one single type,

```
/* Linux kernel source code */
1  struct fib6_walker {
2    struct fib6_info *leaf;
3    ...
4  }

5  struct ipv6_route_iter {
6    struct seq_net_private p;
7    struct fib6_walker w;
8    ...
9  }

10 void *ipv6_route_seq_next(
     ↪ struct seq_file *seq,
     ↪ ... ) {
11   ...
12   struct net *net =
       ↪ seq_file_net(seq);
13   struct ipv6_route_iter
       ↪ *iter = seq->private;
14   ...
15   if (r > 0)
16     return iter->w.leaf;
17   ...
18 }
```

```
; the corresponding LLVM-14 IR
19 define i8* @ipv6_route_seq_next(%struct.seq_file* %seq, ...) {
20   ...
21   %private = gep %struct.seq_file, %struct.seq_file* %seq, i64 0, i32 11
22   %0 = bitcast i8** %private to %struct.seq_net_private**
23   %1 = load %struct.seq_net_private*, %struct.seq_net_private** %0
24   ...
25   %leaf = gep %struct.seq_net_private, %struct.seq_net_private* %1, i64 5
26   %28 = bitcast %struct.seq_net_private* %leaf to i8**
27   %29 = load i8*, i8** %28
28   ...
29 }

; the corresponding LLVM-16 IR
30 !1 = !{!"any pointer", ... }    ; TBAA generic pointer type
31 !2 = !{!3, !1, i64 112 }        ; TBAA access tag
32 !3 = !{!"seq_file", ..., !1, i64 112 }
33 define ptr @ipv6_route_seq_next(ptr %seq, ...) {
34   ...
35   %private = gep %struct.seq_file, ptr %seq, i64 0, i32 11
36   %0 = load ptr, ptr %private, !tbaa !2
37   ...
38   %leaf = gep %struct.ipv6_route_iter, ptr %0, i64 0, i32 1, i32 3
39   %12 = load ptr, ptr %leaf
40   ...
41 }
```

Fig. 3. An example shows that typed-pointer IR uses one type from multiple candidates as a representative.

`ptr` is defined with a general type, i.e., `void *` in the source code and `i8 *` in the IR. If analyzers search for objects of type `struct io_rings *` or `struct io_uring_sqe *`, they cannot determine that this pointer also refers to the desired objects. A conservative solution is to treat the general-type pointer as pointing to any object, leading to many false positives. Furthermore, LLVM IR does not distinguish between `char *` and `void *` and translates them into the same IR type as `i8 *`, so more pointers will have false positives in this solution.

Second, in other cases, one concrete type is selected as a representative out of the multiple options due to the single-type design. This choice leads to missing other types (i.e., false negatives) if the pointer corresponds to multiple types. For example, a struct-type object shares the same address as its first field. An IR variable may chaotically use either of the two types without an explicit type cast. As shown in Figure 3 (line *16*), the return value is loaded from `iter->w.leaf`. However, in the compiled IR, the IR pointer corresponding to `iter` does not have the struct type (`ipv6_route_iter`). Instead, it has the type of its first field, i.e., `%1` at line *23*. When accessing `leaf` in another field (with struct type `fib6_walker`) in the parent struct type `ipv6_route_iter`, IR directly uses the first field's type at line *25* in Figure 3. If the first field's type is considered as the single type for `%1`, the type of `leaf` will be inaccurate. However, if we can infer that `%1` has multiple types to include `ipv6_route_iter`, `leaf` will be correctly inferred with type `fib6_info`.

Lenient opaque pointers. In LLVM 15 and later, the opaque pointer (`ptr`) is introduced, causing all pointers to discard the concrete pointee types, i.e., a kind of generic pointer, as shown in Figure 1. However, many analyses rely on type information for various analysis tasks and purposes [5, 25, 30–32, 35, 42, 48, 50, 57]. Opaque pointers prevent type-based analyses from directly obtaining types from LLVM IR. For example, concrete types of function pointers at indirect call sites are critical to building type-based call graphs, but they cannot be extracted from LLVM IR. Besides, many security-critical objects are stored in memory and accessed via pointers, and collecting these objects typically relies on matching pointee types, rendering this approach ineffective.

The importance of pointee types in static analysis has sparked interest in why the LLVM community discontinued their maintenance. Primarily, a single pointee type often misrepresents actual memory types, causing three kinds of technical difficulties: First, developers must handle the types carefully through back-and-forth type casts between different pointee types. As shown in Figure 3, the LLVM IR with opaque pointers is clearer than the IR with typed pointers. Second, maintaining

pointee types is error-prone and complex, and abandoning them simplifies implementation. Finally, the existing typed pointers have hindered rather than supported LLVM's optimizations. The optimizers often face challenges when attempting to traverse LLVM's struct types and comprehend the underlying memory offsets. To sum up, it has become evident within the community that typed pointers impede rather than aid LLVM's development [38].

Limitations of existing methods for type restoration. Other methods can restore concrete pointer types after enabling opaque pointers in LLVM IR, but they only cover a small portion of instructions and are inadequate for type-based analysis. Existing methods include (1) extracting types by querying Type-based Alias Analysis (TBAA) metadata, and (2) matching the names of IR variables and the source code. Additionally, both methods retain the single-type design, which cannot display multiple concrete types.

First, our experiments on the Linux kernel show that Type-based Alias Analysis (TBAA) covers only 3.92% of IR variables, which is too low to conduct type-based analyses. (The results of other programs are listed in Table 2.) Specifically, TBAA metadata [28] embeds memory types only for the pointers in `load` and `store` instructions, while it does not support other kinds of instructions, as shown in Figure 3 at line *36*. We explain how to calculate this number step by step. (1) A part of `load` and `store` IR instructions include TBAA metadata, which account for 12.46% among all IR instructions. (2) Among the instructions with TBAA metadata, only 89.4% specify the memory types. In other words, 10.6% are marked as `any pointer` (a generic pointer type). (3) Finally, multiple TBAA metadata can describe the type of the same IR variable. After removing duplicate IR variables, 3.92% of IR variables have concrete pointer types by querying TBAA metadata.

Second, the concrete types could be restored by mapping IR variables to their corresponding source code based on name matching, but this approach only covers 2.04% of IR variables in the Linux kernel. Only a small portion of named variables have explicit counterparts in the source code. Many derived IR variables, created to support the SSA format, lack explicit mappings to source-code variables. Their types are critical to static analysis that exploits the SSA feature. Without inferring their types, the analysis would be no different from solely analyzing the source code.

In summary, with the introduction of opaque pointers in the latest and future Clang/LLVM compilers, there is an urgent need for static analysis to keep pace with compiler advancements. Since existing methods are insufficient to restore concrete types of opaque pointers, this work aims to solve this problem. Given the limitations of the single-type design, our solution leverages a multiple-type design to infer types for a broader range of IR variables.

3.2 Multiple-Type Design

To avoid being too restrictive or too lenient, it is necessary to provide a small set of multiple possible types for an IR variable, known as the ***multiple-type design***. The multiple types are collected into a type-alias set. Based on type-alias sets, we introduce a new kind of analysis, ***type-alias analysis***. This analysis identifies accurate and precise types that represent underlying memory types for pointer variables. Our type-alias analysis differs from the widely known type-based alias analysis. In type-alias analysis, two types are considered aliased if they belong to the same type-alias set of a variable, meaning the actual memory types can be any of them. In contrast, type-based alias analysis uses type information to determine whether two pointers are aliases (i.e., they point to the same memory location). Furthermore, our type-alias analysis enhances the previous static analysis methods by providing more accurate type sets to help determine whether pointers are aliased.

The type-alias analysis is non-trivial to achieve. Back to the example in Figure 2, the pointer at line *41* can be resolved with a type-alias set including two types, i.e., `struct io_uring_sqe **` and `struct io_rings **`. This resolution is very challenging when dealing with massive opaque

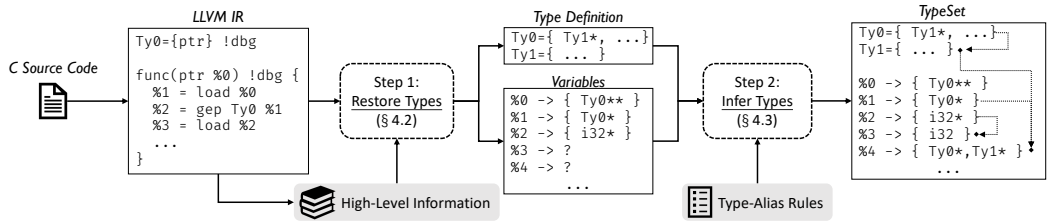


Fig. 4. Overview of our framework, TypeCopilot.

pointers. First, a lot of type information disappears, decreasing information entropy. All pointer-cast instructions are removed, such as `%5` and `%6`. Starting from the beginning, all parameters in Figure 2 have the same type `ptr`. The following load instruction is changed to `%4 = load ptr, ptr %3`. Then, two pointers pointing to two fields at line *33* and line *37*, i.e., `rings` and `se_sqes`, also become opaque. Second, more variables in opaque-pointer LLVM IR have multiple types than those in typed-pointer LLVM IR. This is mainly because opaque pointers cause many instructions to be removed or merged, e.g., `cast`. IR can use fewer variables to achieve the same functionality.

To achieve type-alias analysis, we propose a new framework to support the multiple-type design, TypeCopilot. It aims to provide accurate type-alias sets to improve static analyses. The core idea of TypeCopilot is to bypass the limitations of the single-type design. Rather than modifying the original type system of LLVM IR, it maintains separate data structures, i.e., type-alias sets, to place all possible accurate types, serving as a copilot for various static analyses. We believe TypeCopilot is essential and timely, providing substantial benefits for current and future research.

4 Design of TypeCopilot

TypeCopilot provides types that should be consistent with high-level languages, including type definitions and type usages. Besides, pointer types should reflect all possible types of underlying pointee memory. As opaque pointers have already discarded pointee types, it is necessary to recover them while supporting the multiple-type design. Our key insight is that *we can restore the accurate types for the IR variables that only have one single type, serving as the starting points. Then, we can infer type-alias sets for more IR variables by tracing propagation from the starting points.* Achieving TypeCopilot's goal is non-trivial due to two major challenges:

Challenge 1: For IR variables that act as starting points, obtaining their single types with accurate pointee information is challenging, especially when ensuring that there is enough information to infer the type-alias sets of other variables. The introduction of opaque pointers, which removes all pointee types (causing a reduction of information entropy), exacerbates this challenge. The quality of the starting points is essential because they can affect the inferred type-alias sets.

Challenge 2: Starting from variables with single types, analyzing their propagation across various IR instructions is difficult. To fulfill the requirements of static analysis, types of massive opaque pointers without pointee information need to be inferred. Particularly in the context where opaque pointers make more variables have multiple types, inferring concrete types is challenging.

4.1 Design Overview

To tackle Challenge 1, we propose out-of-band type source restoration in §4.2, which leverages reliable type information from a high-level language to infer IR variables with a single type. We refer to them as ***type sources***, which can be accessed outside of LLVM IR's type system. Specifically, we restore the types of declared IR variables, serving as the starting points. The reasons are as

follows: (1) The declared IR variables have clear and direct mappings with the variables in the source code. We can restore their accurate and single types by obtaining their types from the source code. These variables correspond to the definition sites of global variables, local objects, and heap objects. (2) All pointee memory objects are created at declared IR variables. Therefore, they can cover all starting points to help trace propagation.

To address Challenge 2, we propose a new technique called *TypeInfer* in §4.3, an alias-rule-based type inference method. It is the first inference mechanism for the multiple-type design, which uses type-alias rules to formalize the type propagation across various IR instructions. *TypeInfer* is specifically designed to capture all possible underlying types to build the type-alias sets. This inference can infer the types of derived variables in IR instructions. As most IR variables are derived to support SSA forms and guarantee complex functionality, they may have multiple types, highlighting the need for type-alias rules.

The workflow of TypeCopilot is depicted in Figure 4, comprising two key steps: type restoration and type inference. Following these two key steps, TypeCopilot generates the type-alias sets for IR variables, which can be queried by other static analyses. It is easy to integrate TypeCopilot with existing tools to facilitate their type analysis - developers simply need to replace the original type-requiring APIs with TypeCopilot-provided ones. Once TypeCopilot completes building the type-alias sets, these tools can be executed seamlessly.

Applicable scenario. We assume that users can fully access the source code written in C programming language and analyze it using any compilation options. This scenario is common and practical, as numerous works [3, 21, 23, 25, 31, 33, 42, 52, 55, 57, 60] share this scenario, particularly for analyzing open-source infrastructures and large-scale programs such as the Linux kernel.

4.2 Restoring Types From Type Sources

Before restoring the types of declared variables, TypeCopilot needs to guarantee that struct types have accurate definitions with pointer fields with concrete types. Therefore, the restoration process consists of two parts: type definitions and binding types for declared variables.

Type definition restoration. TypeCopilot utilizes type definitions from source code programmed in C language instead of the translated ones in IR's type system. It can easily collect them because the source-code type definitions are independent of LLVM IR syntax. Although opaque pointers cause all fields in structure definitions in IR to lose pointee types, TypeCopilot can recover them. As shown in Figure 1, TypeCopilot restores the concrete types (`cred *`) of the three fields.

Type restoration of declared variables. All IR variables are divided into two categories: pointer variables related to allocated memory and non-pointer variables. In a high-level language, many variables are in memory, such as global variables in data/bss sections and local variables in the stack that are created by calling `alloca` in LLVM IR. On the contrary, almost all IR variables are single-value types [29], including primitive and pointer types, which can be easily lowered to registers in machine instructions. The only exception is that an IR variable can be a literal struct type, which is restricted to use in function parameters or return values. We can also restore their types, which will be discussed in §5.2.3. Regarding the primitive types in IR, they are directly and simply translated, e.g., `int` to `i32`. These primitive types are embedded in IR instructions, e.g., `ret i32 0`. We assume they also reflect high-level language types and treat them as type sources.

For pointer types, their pointee objects reside in memory and originate from allocation sites. These sites include global variables, local variables in the stack, and heap data. Additionally, we treat function parameters as type sources, and the details are as follows:

(1) Global variables and local variables. For global and local variables in the stack, their types in IR have one more pointer indirection level than the corresponding ones in the source code. For

example, a global variable with a non-pointer type in the source code, e.g., `int`, becomes a pointer in IR (`i32 *`), and `int *` becomes `i32 **`. This pattern holds for all global and stack variables. These variables have clear one-to-one mappings from the source code to IR. As a result, TypeCopilot can restore these IR variables' types by searching their corresponding types in the source code.

Additionally, certain local variables are directly lowered into IR variables instead of being allocated in the stack stored in memory, i.e., not calling `alloca`. We treat these local variables as declared variables as well because of their one-to-one mappings with source-code variables.

(2) Heap data. An object is allocated in the heap by calling specific interfaces, such as `malloc`. These interfaces are usually called with an argument that specifies the type's size that is being allocated. It returns the address of the heap object in the memory allocated with the type, which is stored in an IR variable. TypeCopilot interprets this IR variable's type as a pointer type whose pointee type is the allocated memory type.

(3) Function parameters. In theory, only tracing the allocated memory with points-to analysis can recover all pointers' types. However, precise points-to analysis for large-scale programs is always an open question [6, 7, 44, 51]. We assume the parameters' types are accurate and can be used as type sources. Specifically, if the parameters are pointers, the pointee types reflect the underlying memory types. This assumption is true in most cases. Our analysis shows that over 90% of the arguments of call instructions share the same types as the corresponding parameters of the functions in the Linux kernel. For example, as shown in Figure 1, a function defined in the source code contains a parameter of `cred *`. We can integrate the accurate type `cred *` into the IR parameter at line *10*, which originally has the `ptr` type.

4.3 Type-Alias Algorithm

By tracing the complex propagation starting from declared variables with reliable type sources, *TypeInfer* can infer more variable types and populate the type-alias sets. To achieve this, we first formalize the type flows in LLVM IR instructions into a set of *type-alias rules* based on multiple-type design. For completeness, our rules cover all IR instructions consistent with other LLVM analysis works [13, 17, 51]. To implement these rules, we propose a new technique named *Alias-Rule-based Type Inference (short as TypeInfer). TypeInfer* maintains standalone type-alias sets for IR variables without modifying IR, facilitating previous type analyses to query types from these sets.

4.3.1 The Novelty of TypeInfer. A type inference method comprises two essential components: a language-specific type organization and a core type inference. We elaborate on the novelty of *TypeInfer* through comparison with existing approaches from the two perspectives.

Type organization designed for IR with opaque pointers. To the best of our knowledge, no previous type inference can directly apply to LLVM IR with opaque pointers due to the language-specific type organization. In this paper, the organization includes type lattices and type binding for variables. Existing type inference approaches are designed for other scenarios, such as inferring types for assembly binary and dynamic languages. However, their scopes do not include LLVM IR with opaque pointers. *TypeInfer* specially designs type organization for this scenario. The details of type lattices and type binding of *TypeInfer* will be illustrated as follows, respectively.

(1) *TypeInfer* designs the type lattice in the scenario of inferring the source-code types of opaque pointers in LLVM IR. It primarily adheres to the type lattice of the C language, while incorporating IR types from two perspectives. For pointers, we establish a hierarchical relationship $ptr <: void* <: type*$, where opaque pointers occupy the bottom tier, followed by `void *`, and concrete-typed pointers at the top. The subtype relation is represented as $<:$. Specifically, a type T_a is a subtype of T_b, written as $T_a <: T_b$, if and only if any term of type T_a can be safely used in a context where a term of type T_b is expected. As for integers, we define relationships between

```
/* Linux kernel source code */
1 struct sock {
2   ...
3   void *sk_security;
4   ...
5 }

6 void sock_copy(struct sock *nsk, ...) {
7   void *sptr = nsk->sk_security;
8   ...
9 }

10 int selinux_sk_alloc_security(struct sock *sk, ...) {
11   struct sk_security_struct *sksec;
12   sksec = kzalloc(sizeof(*sksec), priority);
13   ...
14   sk->sk_security = sksec;
15 }

16 int smack_sk_alloc_security(struct sock *sk, ...) {
17   struct socket_smack *ssp;
18   ...
19   sk->sk_security = ssp;
20 }
```

Fig. 5. *TypeInfer* uses the type-based access path to resolve the concrete types of generic pointers.

IR integer types (I) and their C language counterparts (T) of equivalent bit-width as $I <: T$, e.g., $i32 <: int$.

The type lattices in existing type inferences vary depending on the language analyzed, which is not applicable to this paper. For comparison, we use the type lattices from binary analyses because they are closely related to LLVM IR. These works focus on lifting assembly code to IR with reconstructed types [4, 8, 11, 15, 20, 39, 54, 58]. Their type lattices are more complicated due to the ambiguity between pointers and integers stored in registers, such as $reg32 <: ptr$ and $reg32 <: num32 <: uint32$ [20]. In contrast, LLVM IR explicitly distinguishes between pointers and integers, which facilitates more precise type inference for pointer types.

(2) *TypeInfer* infers concrete types for generic pointers, allowing variables to bind multiple types simultaneously. This contrasts with existing inference approaches used to generate LLVM IR with inferred IR types, which are in conformance with the single-type design. Specifically, *TypeInfer* is designed to resolve opaque pointers (`ptr`, a specialized form of generic pointers) and the pointers with `void *` type. However, existing works default to generic pointer types (`void *`) when variables are inferred to have multiple types [15]. Besides, their inference process stops after inferring the generic types because they assume `void *` is aligned with the source code already. On the contrary, *TypeInfer* continues the inference process to resolve the concrete pointer types until it encounters declared variables or variables with inferred types.

Type inference with type-based access paths. We propose type-based access paths to precisely and efficiently resolve concrete types for generic pointers. As data flow can be broken across multiple entries, types can be used to reconnect them. Specifically, if a generic pointer's type cannot be resolved by tracing data flow, *TypeInfer* analyzes the types along its access path (i.e., struct types and field index) and adds the found types to its type-alias set. This inter-procedural analysis continues until it identifies the concrete types from memory allocation sites (as discussed in §4.2) or variables with inferred types. For example, a local variable `sptr` at line *7* in Figure 5 is defined with type `void *`. It is initialized with `nsk->sk_security` whose type is `void *` also. *TypeInfer* uses the type-based access path, `sock, sk_security` in this case, to find which types are assigned to this access path. After analyzing other functions globally, we find `sksec` (line *14*) and `ssp` (line *19*) are assigned to the identical struct type and field. Finally, the type set of `sptr` consists of two types corresponding to the two variables, `struct sk_security_struct *` and `struct socket_smack *`.

Type inference fundamentally depends on tracking memory object propagation — a challenge traditionally addressed by pointer analysis methodologies [4, 15, 58]. Our method is different from existing points-to analyses, and we compare it with two popular analyses below.

(1) Andersen-style analysis [1, 45] computes the points-to sets consisting of the locations where the pointee objects are created, which takes high-performance overhead. Differently, *TypeInfer* computes a type set consisting of all types of the pointee objects, achieving high scalability. Each type

describes a set of locations where objects of this type are created. We choose the type-based method because it can scale to large programs, whereas Andersen-style analysis introduces significant computational complexity. Therefore, Andersen-style analysis is impractical for analyzing large-scale programs [9, 21, 31]. For example, previous work analyzes the time cost of individual global variables in the Linux kernel, and results show that the analysis of about 50% of variables cannot be finished within 8 hours [21].

(2) The difference from k-CFA [41] (a type-based points-to analysis) is that *TypeInfer* leverages type-based access paths to resolve types of generic pointers in LLVM IR. k-CFA [41] is proposed to recover types for higher-order languages and then applies to type-safe object-oriented languages [34, 49]. The basic idea of *TypeInfer* is similar to 0-CFA. However, to be applicable to non-object-oriented and type-unsafe languages (i.e., it allows arbitrary pointer type casts), *TypeInfer* is specialized in type resolving for generic pointers. This is because the generic pointers (opaque pointers and `void *`) are special features in low-level languages. Our analysis leverages type-based access paths to resolve concrete types for them with the scalability of large programs.

4.3.2 Program Representation. The set of all variables V is separated into two subsets. (1) A: contains all possible abstract objects, i.e., address-taken objects as the pointees of pointers. (2) P: contains all IR variables, including local variables (symbols starting with "%") and global variables (symbols starting with "@"). $\mathcal{T}$ is a set that contains all non-pointer types, including non-pointer primitive data types and aggregate data types. The LLVM IR is represented by nine kinds of statements. Specifically, $p \leftarrow \&o$ (AddrOf), $p \leftarrow \&(q \rightarrow i)$ (Field, get the address of a field from a pointer pointing to a struct object), $p \leftarrow (type)q$ (Cast), $p \leftarrow q$ (Copy), $p \leftarrow *q$ (Load), $*p \leftarrow q$ (Store), $r \leftarrow p \otimes q$ (Binary, including arithmetic and logical operations), $r \leftarrow \phi(p, q)$ (Phi), $p \longleftarrow q(r_1, ..., r_n)$ (Call) where $o \in A$; $p, q, r \in P$; and i is an integer index.

We define a type to consist of two parts, (τ, η), where $\tau \in \mathcal{T}$ represents a base non-pointer type, and η is a non-negative integer number representing the pointer indirection level. If the variable is not a pointer, η will be 0, e.g., `struct.cred *` is represented as $(cred, 1)$ and `i32` is represented as $(i32, 0)$. We use tar to represent a mapping that maps a unique IR variable to its valid type-alias set. The rules use $(\tau, \eta) \in tar(p)$ to iterate each type in the set of a variable p.

Type source query. We use $\tau \leftarrow \mathcal{F}()$ to represent querying type sources. $\mathcal{F}()$ is implemented in three different ways, and we will discuss each one separately. First, since all types of allocated memory are preserved, *TypeInfer* can naturally figure out the type of a variable that is initially assigned with the allocated memory's address, as the rule of AddrOf shows. Second, when getting a pointer that points to the `i`-th field, *TypeInfer* refers to the accurate definition that has been restored from the source code. Afterwards, it can find the accurate field type by adding the offset within the referred definition. Third, before merging the two sets in the cast instruction, *TypeInfer* first obtains the type of the destination variable, i.e., p, by querying the type sources.

Adjustment for opaque pointers. *TypeInfer* includes the adjustment for `load` and `store` instructions, represented by $\Delta(p)$. It adjusts the type-alias types by tracking memory objects and applying constraints to ensure the types are compatible with LLVM IR. This is because the opaque pointers cause many instructions to be wiped out, e.g., cast instructions, causing a variable to have multiple types without an explicit cast. For example, we have restored the parameter type, `struct cred *` in Figure 1. We first meet a `load` instruction at line *12*, and its variable's type is supposed to be `struct cred`. By conducting our points-to analysis with constraints of compatibility with LLVM IR (i.e., the variable should be a pointer), TypeCopilot figures out the pointer points to the first field `usage`, and the type of (`%4`) is `atomic_t`. This struct type (`atomic_t`) contains only one integer field in the source code, and it is expanded as `i32` in LLVM IR, allowing it to be used in `icmp` instruction.

Table 1. Accurate type-alias rules trace type flow to infer more IR types based on the type source.

[AddrOf] $\frac{p \overset{AddrOf}{\longleftarrow} o, (\tau, \eta) \overset{Type}{\longleftarrow} \mathcal{F}(o)}{(\tau, \eta+1) \in tar(p)}$	[Cast] $\frac{p \overset{Cast}{\longleftarrow} q, (\tau_p, \eta_p) \overset{Type}{\longleftarrow} \mathcal{F}(p)}{(\tau_p, \eta_p) \in tar(q), tar(q) \subseteq tar(p)}$
[Field] $\frac{p \overset{Field_i}{\longleftarrow} q, (\tau_q, \eta_q) \in tar(q), (\tau_f, \eta_f) \overset{Type}{\longleftarrow} \mathcal{F}(\tau_q, i)}{(\tau_f, \eta_f+1) \in tar(p)}$	[Load] $\frac{p \overset{Load}{\longleftarrow} q, (\tau, \eta) \in tar(q)}{(\tau, \eta-1) \in tar(p), \Delta(p)}$
[Call] $\frac{p \longleftarrow q(r_1, ..., r_n)}{tar(p)(tar(r_1), ..., tar(r_n)) \subseteq tar(q)}$	[Copy] $\frac{p \overset{Copy}{\longleftarrow} q}{tar(q) \subseteq tar(p)}$
[Store] $\frac{p \overset{Store}{\longleftarrow} q, (\tau, \eta) \in tar(q)}{(\tau, \eta+1) \subseteq tar(p), \Delta(p)}$	[Binary] $\frac{r \longleftarrow p \otimes q}{tar(p) \bigcup tar(q) \subseteq tar(r)}$
	[Phi] $\frac{r \longleftarrow \phi(p, q)}{tar(p) \bigcup tar(q) \subseteq tar(r)}$

4.3.3 Rule Statement. Table 1 lists type-alias rules applied to nine kinds of statements. Our analysis realizes these rules in a field-sensitive, flow-insensitive, context-insensitive manner. For scalability, the loop is unfolded once instead of iterating to fixed points. On the other hand, the type-alias set for each variable is computed until it stabilizes to reach a fixed point.

- AddrOf: When a pointer p initially points to an object o, *TypeInfer* obtains o's type from type sources, (τ, η). The type-alias set of the pointer $tar(p)$ is then updated by adding this base type with an increased pointer level, $(\tau, \eta+1)$.
- Field: When taking out the i-th field of struct object whose address is stored in the pointer q and storing the field's address to p, *TypeInfer* derives the type of i-th field τ_f from the type source (i.e., the struct definition in the source code). After adding one pointer layer of the field type, it becomes the type of the variable storing the field address, p. Note that q represents a pointer to a memory object with the base type τ_q, and its pointer indirection layer (η_q) is one in all cases. This constraint helps filter out false positives.
- Cast: q has a type-alias set $tar(q)$, and it is cast to another IR variable p with a different type. *TypeInfer* utilizes the type embedded in the cast instruction to get the destination type (τ_p, η_p), which is added into the set of the source operand, i.e., $tar(q)$. Afterwards, $tar(q)$ is merged with the type-alias set of p. This ensures the type set includes all possible memory types, as the memory address remains unchanged after casting. This rule applies only to LLVM IR with typed pointers, not to opaque pointers, because no cast exists between two pointers.
- Call: When calling a function q that has n arguments $r_1, ..., r_n$ and a return value p, *TypeInfer* organizes them as a function's type. Specifically, $tar(p)$ represents the type-alias set of the return value, and $tar(r_1), \cdots, tar(r_n)$ represent the arguments in order. These types are organized as the function's type, which is put into the type-alias set of this function q.
- Store: When storing q to p, the type-alias set of q will be passed to the set of the pointer p. Similarly, each base type in the set of q remains the same, but the pointer level is increased by one. It also includes the adjustment by points-to analysis, $\Delta(p)$.
- Load: When p loads from a pointer q, the type-alias set of q will update the one of p. The base non-pointer type of each type in the type-alias set of q remains the same. The pointer level is reduced by one. Additional points-to analysis can adjust the set of q by $\Delta(p)$.

Table 2. Programs for evaluation and corresponding ground truth sizes.

Programs	Version	LoC	IR Variables	Ground Truth Size	
				TBAA	CodeQL
Coreutils	9.5	709K	1,087K	17.1K (1.57%)	6.3K (0.58%)
Nginx	1.25.4	164K	141K	5.8K (4.11%)	3.9K (2.77%)
OpenSSL	3.4	744K	95K	1.1K (1.16%)	4.2K (4.42%)
Linux	5.15	24,535K	5,295K	207.7K (3.92%)	108.2K (2.04%)

- Copy: *TypeInfer* puts all elements in the type-alias set of the source variable q into the set of the sink variable p, if there is an assignment between two variables.
- Binary: The variable r is computed by two operands p and q, so its type-alias set takes the union of the two sets of the operands. This merge operation is to avoid missing any true positives.
- Phi: A phi node has multiple incoming values, and the types of r are computed by merging the types of incoming operands.

We use the example in Figure 2 to demonstrate how these rules work on various instructions. First, we start with the variable `ctx` at line *10*, which has a one-to-one mapping with the declared IR variable at line *30*. Figure 2 shows the previous IR version of typed points, while all pointers become `ptr` after introducing opaque pointers, i.e., `%4 = load ptr, ptr %3`. We can obtain this declared variable's type by querying the type source, i.e., (`%4`) has the type `struct io_ring_ctx *`. Afterwards, the types of the two fields at line *33* and line *37* are inferred by looking up the fields in the struct (Field). Finally, the phi node receives the two types and puts them into its type-alias set.

5 Implementation and Evaluation

We built a prototype of TypeCopilot using 2.5 kLOC C++. Our implementation is based on LLVM 14 with typed pointers and 16 with opaque pointers [18]. We ran all experiments on an AMD EPYC 9654 machine running Ubuntu 22.04. We evaluate TypeCopilot on the LLVM IR of various popular C programs. Our evaluation focuses on two key aspects: the accuracy and coverage of TypeCopilot, including distinct evaluations of type restoration and *TypeInfer*. We then present case studies demonstrating how TypeCopilot integrates with type-based analysis tools to enable their effectiveness on LLVM IR with opaque pointers.

Implementation details of TypeCopilot. In addition to primitive types already embedded in IR instructions, we refine our type sources using debug information metadata. Previous works demonstrate that optimizations can disturb the explicit mappings between source code variables and IR variables [14, 22, 61]. Therefore, we disable all optimizations when running TypeCopilot. Debug information contains all type definitions, global variables, and local variables. By traversing `getGlobalVariable()` and `alloca` instructions, one can easily find these variables in IR with one-to-one debug information metadata (`!dbg`). We manually collect heap allocation interfaces and connect their types for IR variables. Finally, we can obtain parameter types using debug information because each function also has an explicitly attached `!dbg`.

5.1 Experiment Setup

Testing program set. To evaluate TypeCopilot, we first build a micro-benchmark containing popular C programs, including both userspace and kernel programs. As listed in Table 2, we choose Coreutils, Nginx, OpenSSL, and the Linux kernel as targets. These programs differ in program types and size, including both user space programs and the fundamental Linux kernel. In particular,

Coreutils contains over 90 separate programs to test different programming patterns. Each program contains a large number of lines of source code and IR variables, aiding in testing the scalability of TypeCopilot and obtaining average coverage and accuracy results with low randomness.

Compilation setting. To run TypeCopilot for each program, we compiled LLVM IR using the compiler's built-in options to attach type sources, i.e., `-g` for debug information. The primitive type sources are embedded in LLVM IR instructions by default without any compile option. From the perspective of evaluation, we need to enable one more compilation option, which is not required to run TypeCopilot. Specifically, `-fstrict-aliasing` preserves variables' names in the source code, which facilitates comparison with ground truth. This option also preserved TBAA (type-based-alias analysis) metadata, which is used as ground truth.

Ground truth building. We construct two ground truth sets to evaluate TypeCopilot's effectiveness in inferring concrete types for generic pointers on LLVM-16 IR. The sizes of the two ground truth sets for different programs are shown in Table 2. (1) We use TBAA metadata [28] as another ground truth that is fully compatible with LLVM IR variables. TBAA metadata preserves type information for alias analysis and is used by dynamic type sanitizers [16], making it an accurate representation of memory types. (2) We build a ground truth using CodeQL [2], which extracts accurate type information from source code into a queryable database. Since LLVM IR variable names partially match source code names, we can map variables between IR and the CodeQL database to obtain their original types.

We evaluated TypeCopilot beyond limited ground-truth sets. This is because the two sets cannot cover all variables, ranging from 0.58% to 4.42%, as shown in Table 2. We argue that creating a complete ground truth for large-scale programs is impractical. To further validate our results, we conduct an experiment: We run TypeCopilot on LLVM-14 IR with typed pointers because the types from LLVM-14 IR are widely used in static analyses. We believe most types in LLVM-14 IR are accurate and check if TypeCopilot can infer them correctly. Acknowledging this limitation, we plan to open-source our implementation, inviting the research community to refine and enhance it.

Baseline construction. After enabling opaque pointers, ***residual types*** are still preserved in LLVM IR to maintain its functionality in any case [27], so we built the baseline by directly handling IR variables that have residual types, without performing type inference. This baseline uses IR variables as the evaluation granularity to assess the number of variables with residual types and the number of residual types that are consistent with the ground truth. We treat all non-pointer IR variables as having residual types, such as those with type `i32`. As for pointer types, residual types are scattered in global variables and several specific instructions, which contain concrete types for opaque pointers `ptr`. However, not all pointer IR variables correspond to residual types. For example, there is residual `i32` in the load instruction at line *12* of Figure 1, indicating the pointer has a type of `i32 *`. Another counter-example is that there is no residual type when the loaded type is also a pointer, `%1 = load ptr, ptr %2`.

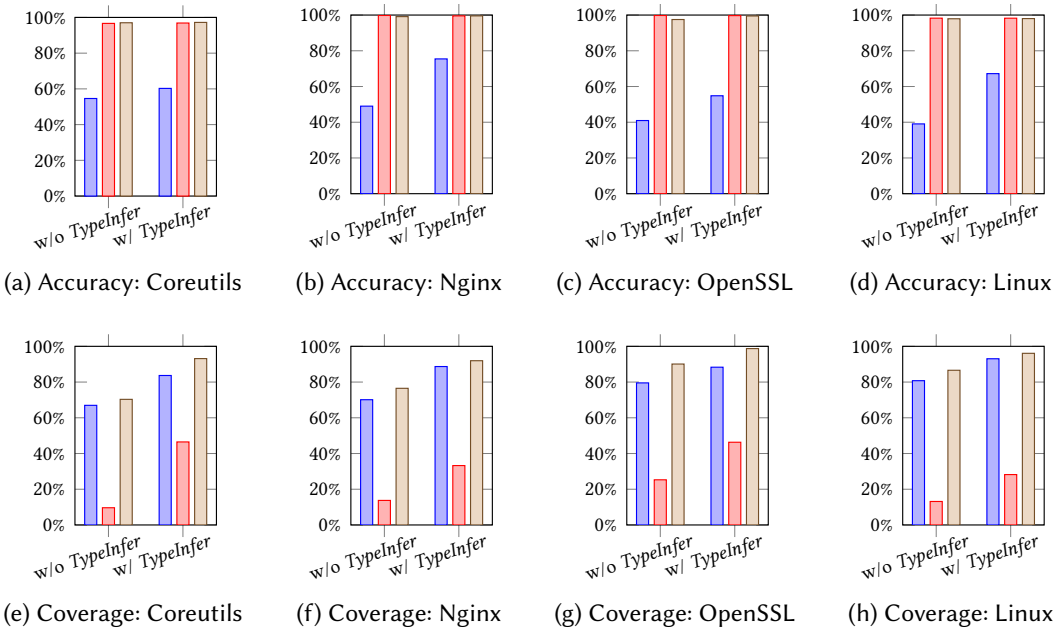
5.2 Evaluation on the Effectiveness of TypeCopilot

We evaluate TypeCopilot on LLVM IR with opaque pointers to thoroughly test its effectiveness when all pointee types are lost. This evaluation is critical for recovering types in modern LLVM compilers, enabling type-based analyses. The effectiveness is specifically manifested in counting how many IR variables are inferred with accurate types. This section includes the effectiveness evaluation of two major parts of TypeCopilot: type-source restoration and type-alias rules.

Coverage definition. We define the coverage percentage as $\frac{Infer}{All}$, where $Infer$ represents the number of IR variables whose types are inferred, and All represents the number of all IR variables. Specifically, if a pointer is still an opaque pointer without concrete pointee types, we consider it

Table 3. Accuracy and coverage comparison on TBAA ground truth and CodeQL ground truth between the baseline and TypeCopilot in LLVM IR with opaque pointers. In particular, pp represents percentage points.

Programs	TBAA Ground Truth		CodeQL Ground Truth		Coverage	
	Baseline	TypeCopilot	Baseline	TypeCopilot	Baseline	TypeCopilot
Coreutils	16.08%	94.97% (78.89 pp ↑)	54.62%	97.21% (42.59 pp ↑)	66.97%	93.14% (26.17 pp ↑)
Nginx	4.17%	94.15% (89.98 pp ↑)	49.00%	99.54% (50.54 pp ↑)	70.12%	91.95% (21.83 pp ↑)
OpenSSL	32.06%	97.65% (65.58 pp ↑)	40.95%	99.49% (58.54 pp ↑)	79.53%	98.74% (19.21 pp ↑)
Linux	6.14%	90.16% (84.02 pp ↑)	39.06%	98.03% (58.97 pp ↑)	80.77%	96.09% (15.32 pp ↑)
Average	14.61%	94.23% (79.62 pp ↑)	45.91%	98.57% (52.66 pp ↑)	74.38%	94.98% (20.60 pp ↑)



(a) Accuracy: Coreutils (b) Accuracy: Nginx (c) Accuracy: OpenSSL (d) Accuracy: Linux

(e) Coverage: Coreutils (f) Coverage: Nginx (g) Coverage: OpenSSL (h) Coverage: Linux

Fig. 6. Accuracy and coverage of different type sources without and with *TypeInfer*.

uncovered. As for non-pointer IR variables, the inferred types can be from the type systems of the C language or LLVM IR, such as integers represented as `unsigned int` or `i32`.

Accuracy definition. We define accuracy percentage as $\frac{Infer\_Acc}{GroundTruth}$, where $Infer\_Acc$ represents the number of variables with accurate inferred types, and $GroundTruth$ represents the number of IR variables that correspond to ground truths. We consider a variable to have an accurate type if its inferred type-alias set contains one type that matches the ground truth.

Overall comparison with baseline. As shown in Table 3, TypeCopilot achieves an average accuracy of 94.23% (under TBAA ground truth) or 98.57% (under CodeQL ground truth) and 94.98% coverage. Compared to the baseline, TypeCopilot significantly improves both accuracy

and coverage across all tested programs. Specifically, accuracy shows 79.62 percentage points (pp for short) improvement using TBAA ground truth or 52.66 pp increase using CodeQL ground truth. The baseline shows lower accuracy on the TBAA ground truth set compared to CodeQL because TBAA contains a higher proportion of pointer-typed variables. The baseline leaves many of these as unresolved opaque pointers, which are counted as inaccurate matches. Regarding coverage, TypeCopilot improves by 20.60 pp compared to the baseline.

5.2.1 Effectiveness of Different Type Sources. To comprehensively compare the effectiveness of our type sources, we evaluate other possible type sources in LLVM IR that are accessible with opaque pointers enabled. (1) Residual types: we collect all types obtained through migration instructions [27], which form our baseline. (2) Debug information: we directly and separately obtain types from debug information without the type-source integration process restoration described in §4.2. Specifically, we collect these by traversing all `llvm.dbg.` call instructions and compare them with the type sources in TypeCopilot. As shown in Figure 6, we present the accuracy and coverage of different type sources both without and with *TypeInfer* inference.

Accuracy. In most cases, our type restoration achieves high accuracy, over 97%. Compared with residual types, debug information achieves higher accuracy as it is specifically designed to correlate LLVM IR with source code. The accuracy loss mainly stems from false types of heap objects, such as when allocating an object with a struct type whose last field is an array with a dynamically determined size. On the contrary, residual types generally show the lowest accuracy, which is expected due to missing pointee types.

Coverage. Overall, our type restoration achieves 70.32% to 90.09% coverage, surpassing both residual types and debug information across all four tested programs. Specifically, the coverage of debug information ranges from 9.64% to 25.28%. Residual types show higher coverage than debug information, ranging from 66.97% to 80.77%. The effectiveness of the final results with inferred types is our most important metric, discussed in §5.2.2. In summary, our results show that the inference rules perform better when using our type sources compared to the other sources.

5.2.2 Effectiveness of TypeInfer. To demonstrate the effectiveness of *TypeInfer*, we evaluate accuracy and coverage against the programs listed in Figure 6. We also apply this inference to other type sources to thoroughly test its inference capabilities.

Accuracy. Data shows that *TypeInfer* can lift residual types' accuracy in all the test programs. This is because residual types improve in accuracy after inference, as accurate types propagated from global and local memory are obtained. Note that LLVM provides APIs to access these types, which are correct in most cases. Debug information shows high accuracy across all programs with or without inference. While with debug information and our type restoration, *TypeInfer* incurs a tiny decline in accuracy. However, we argue that such a decline is caused by our conservative ground truth matching and is negligible, more details are discussed later.

In addition, we evaluate the size of the type-alias sets after inference, which serves as auxiliary proof of false positives. We treat a variable type as accurate if it belongs to our built type-alias sets. We calculate the set size to evaluate the false positives. After resolving typedef macros (e.g., redefining a new name with an existing struct) and eliminating type aliases, the average size of type sets ranges from 1.18 to 1.22, and 83.7% – 87.9% variables have only one type in the set.

To evaluate precision, we categorize the root causes of multiple types. First, among all variables with multiple types, 61% are caused by multiple formats of primitive types, including different bit-widths and type redefinitions, such as `int`, `unsigned`, and `i64`. These primitive types are more commonly used as pointee types in LLVM IR than in the source code, as all global and local variables need to add one pointer indirection layer when translating to IR. Many operations between two

kinds of primitive types exist, such as `(int) a == (unsigned) b`, causing multiple types. Second, we find 32% of variables correspond to the two kinds of representations of multiple types mentioned in §3.1: 21% correspond to the generic pointers; 11% correspond to indiscriminate usages between struct types and their first field's type. We manually checked 20 cases and found TypeCopilot inferred multiple types as expected. The remaining ones are caused by complicated propagation across massive instructions, which can vary from case to case and may have false positives.

To evaluate the recall rate, we investigate whether the inferred single type can cover all true positives. Since obtaining proper ground truth for multiple types is challenging, we find a method that samples cases using TBAA metadata. As discussed before, TBAA uses `any pointer` in some cases. After our investigation, this generic pointer indicates that the variable has multiple types or its type has two or more pointer indirection levels. Therefore, we first exclude the cases of multiple indirection levels. Afterwards, we collect the variables that have a single inferred type and TBAA ground truths. Among them, 2% single-type variables correspond to `any pointer`. We manually checked several cases and found that TypeCopilot inferred types accurately as expected under the assumption that function parameters' types reflect underlying memory types.

Coverage. The goal of *TypeInfer* is to recover the types for derived variables based on the propagation of variables with type sources. As depicted in the second row of Figure 6, all the type sources significantly increase coverage using *TypeInfer*, with improvements ranging from 8.65 pp to 36.83 pp. After inference, residual types cover over 83% of the variables, while the coverage of debug information ranges from 28.21% to 46.48% in the tested programs.

Discussion of accuracy and coverage loss. Our current prototype of TypeCopilot cannot achieve 100% accuracy and coverage on several ground truth sets. Based on our manual investigation, there are three main reasons: First, types and variables from external libraries affect the evaluation. For example, Coreutils relies on `timespec` from glibc. However, such structure is not visible for LLVM IR, causing all the type propagations with `timespec` to be unavailable. Second, our judgment criteria for accurate types are conservative. For example, the two types are not matched if their bit widths differ, e.g. a `bool` may be transformed into `i32` implicitly. But we do not consider them as the same types. Third, TBAA provides less precise types compared to TypeCopilot in some cases, causing accuracy loss. For example, when loading from a pointer towards a field of type `int` within a struct, TBAA uses less precise type `int` to describe this memory instead of using the whole struct type. We pushed our limits to achieve higher accuracy by investigating inaccurate cases. We believe existing analyses can benefit from TypeCopilot.

5.2.3 Comparison With Typed-Pointer IR. To further prove the accuracy, we run TypeCopilot on LLVM-14 IR with typed pointers. For each IR variable, if its type in LLVM-14 IR matches an inferred type in its type-alias set, the variable is considered to have a consistent type. The results show that 20.32% IR variables have inconsistent types. The inconsistency is conservative, e.g., deciding two integers are different if they have different bit widths. If solely comparing pointer types without considering integers, there will be 17.18% inconsistency.

We randomly select 30 inconsistency cases and manually investigate the root cause, as manually checking a large absolute number of variables is not possible. We found that these cases resulted from the inconsistencies between LLVM-14 IR and the source code. The compilation process mainly focuses on the program's functionality during translation, so LLVM IR does not have to maintain consistency to preserve semantics. On the contrary, TypeCopilot bypasses this translation step and directly collects type information in the source code. We did not find these inconsistencies in the TypeCopilot result. In the following, we discuss three inconsistencies in built-in LLVM IR.

Inconsistency 1: Duplicated type definitions. LLVM IR contains duplicated definitions for an identical struct type. Figure 7 shows an example that is raised by inconsistent field types. First,

```
/* Linux kernel source code */
1 struct fs_context_operations {
2   void (*free)(struct fs_context *fc);
3   ...
4   int (*get_tree)(struct fs_context *fc);
5   int (*reconfigure)(struct fs_context *fc);
6   ...
7 };

8 struct cred {
9   ...
10  kuid_t uid;  /* real UID of the task */
11  ...
12  struct user_namespace *user_ns;
13  ...
14 } __randomize_layout;
```

```
; the corresponding LLVM IR
15 %struct.fs_context_operations.97981 = type { void
   ↪ (%struct.fs_context.98471*)*, ..., {}*, {}* }

; Literal field types and missing kuid_t
16 %struct.cred = type {..., %struct.atomic_t, ...,
   ↪ %struct.user_namespace*, ...}
17 %struct.cred.65136 = type {..., %struct.atomic_t, ...,
   ↪ {..., [4 x i64]}*, ...}

; Duplicate struct definitions
18 %struct.task_struct = type {..., %struct.cred*,
   ↪ %struct.cred*, %struct.cred*, ...}
19 %struct.task_struct.79626 = type {..., %struct.cred.65136*,
   ↪ %struct.cred.65136*, %struct.cred.65136*, ...}
```

Fig. 7. Duplicate type definitions caused by inconsistent struct layouts, recursively affecting other definitions.

```
/* Linux kernel source code */
1 struct open_how build_open_how(int flags,
  ↪ umode_t mode) {...}

3 typedef struct { uid_t val; } kuid_t;
4 int device_change_owner(struct device *dev,
  ↪ kuid_t kuid, kgid_t kgid) {...}

5 typedef __s64 time64_t;
6 struct timespec64 {
7   time64_t  tv_sec;
8   long    tv_nsec;
9 };

10 struct timespec64 ns_to_timespec64(s64 nsec)
   ↪ {...}
```

```
; the cprresponding LLVM IR
12 define void @build_open_how(%struct.task_cputime*
   ↪ %agg.result, i32 %flags, i16 %mode) {...}

13 define i32 @device_change_owner(%struct.device* %dev,
   ↪ i32 %kuid.coerce, i32 %kgid.coerce) {...}

14 define { i64, i64 } @ns_to_timespec64(i64 %nsec)
   ↪ {...}
```

Fig. 8. The struct types are coerced to other types when using them for IR variables.

`{}*` (line *15*) replaces the last two fields in the definition of `fs_context_operations` (line *4* and line *5*). Second, LLVM IR contains two definitions of `cred` at line *16* and *17* because field `user_ns` is defined with different types. Finally, multiple definitions of `cred` lead to `task_struct` being redefined multiple times, as it contains three fields of type `cred *`, as shown at line *18* and *19* in Figure 7.

In the Linux kernel, we found that the LLVM IR contains 46,957 struct type definitions, of which 40,792 (86.9%) are duplicates. After deduplication, they correspond to 6,165 unique struct definitions in LLVM IR, while 1,317 have duplicated instances in IR. We analyze the root causes of such inconsistency. Generally, a single struct type can span multiple compiling units (e.g., .c files). Its layouts may vary when these files are compiled into their corresponding IR. These variations typically arise when fields of pointer types are inconsistent across different IR files. As all opaque pointers share the same type, this inconsistency is solved in LLVM IR's new version.

Inconsistency 2: Missing type definitions. The second inconsistency is raised because type definitions can be completely wiped out due to optimizations or specific implementations. This inconsistency is also common and exists both in typed-pointer IR and opaque-pointer IR. For instance, as shown at lines *16* and *17* of Figure 7, `kuid_t` type is totally replaced by `atomic_t`. As mentioned before, the Linux kernel's LLVM-14 IR contains 6,124 structs (after deduplication). However, these structs correspond to 9,156 structs in the source code, and 3032 (33.12%) struct types are missing. The situation is even worse in LLVM-16 IR, where opaque pointers are enabled. Only 4,784 structs are in the IR, and 4,372 (47.75%) structs are missing.

The reasons for these missing definitions are multifaceted. First, a struct type may be coerced into other types (more details in inconsistency 3). As a result, its definition becomes redundant and is removed if the type is not used elsewhere. Second, a linker can cause a type to be replaced by

Table 4. Indirect call targets of MLTA with TypeCopilot on LLVM-14 and LLVM-16.

LLVM Version	**LLVM-16**			**LLVM-14**	
MLTA Integration	Baseline	Modified	TypeCopilot	Original	TypeCopilot
# iCall	15,643			15,667	
Avg. Target	33,791	1,141	16.9	16.4	18.6

another with the same layout. Since LLVM IR only needs to retain one definition per unique layout, other identical layouts are removed to improve performance without affecting the compilation. All opaque pointers share the same type, resulting in more struct types having the same layouts.

Inconsistency 3: Inaccurate type usages. If variables are declared using inaccurate types, there will be an inaccuracy in type usage. Naturally, the previously mentioned inaccurate type definitions can contribute to this inconsistency. Moreover, even if all type definitions are accurate, the IR still encounters issues using inaccurate types for variable declarations.

Figure 8 shows multiple examples of inaccurate type usages. First, the return value with struct `open_how` of the function `build_open_how` is compiled to `void` (line *12* of Figure 8). In contrast, LLVM IR declared the return value as the first parameter. Second, when small-sized struct types are used to declare IR variables, they are split into (multiple) fields or converted to literal structs that are deprived of names. At line *4*, both `kgid_t` and `kuid_t` only contain a 32-bit integer field, and they are directly coerced into 32-bit integers, losing semantics. The type-matching-based analysis will fail to find the critical data of `kuid_t`. Finally, the type of the return value at line *10* is `struct timespec64` in a high-level language; however, it becomes literal (line *14*) in IR, causing the type inconsistency.

5.3 Case Study: Integration With MLTA

To demonstrate how type-based analysis tools can benefit from our framework, we integrate TypeCopilot with MLTA [31]. We conduct five experiments. Among them, three on LLVM-16 and two on LLVM-14, with results shown in Table 4.

(1) Baseline: We port MLTA to LLVM 16 with opaque pointers enabled. Due to missing pointee types, the default `getFunctionTy()` API only returns `ptr` type, which could point to any function. This leads to an excessive number of average call targets: 33,791 per indirect call (short for iCall).

(2) Modified: We modify MLTA to leverage residual types in the current type system. Although it matches function signatures based on the original type system, parameters lacking pointee types still result in numerous false positives (1,140 per iCall).

(3) TypeCopilot on LLVM-16: We incorporate TypeCopilot as a plugin within MLTA's pass pipeline. By restoring more precise types, TypeCopilot-enabled MLTA on LLVM-16 successfully reduces the average number of call targets to 16.9.

(4) Original: This represents the original MLTA implementation on LLVM 14, which is used to compare with our integration. The original implementation has 16.4 average call targets. The comparison shows that TypeCopilot-enabled MLTA on LLVM-16 has slightly more average call targets than the original implementation (16.9 vs 16.4).

(5) TypeCopilot on LLVM-14: We further evaluate TypeCopilot-enabled MLTA on LLVM-14. We find it averages 18.6 call targets compared to the original's 16.4, or 16.9 on LLVM-16. This slight increase is acceptable for CFI usage and stems from our multiple-type design.

The increase in average targets with TypeCopilot is primarily due to three factors. First, following Table 1, we bind functions to multiple types by adding types at call sites (Call rule), which enlarges function type sets. Second, indirect call matching now involves comparing sets of types rather

than single types, and we consider parameters and arguments matched if their type sets overlap, naturally leading to more potential matches. Finally, LLVM-14 IR contains pointee types for all pointers, which increases the type-matching opportunities. For example, LLVM-14 IR has unique cast instructions between two typed pointers, and our Cast rule can enlarge type-alias sets.

6 Related Work

Security static analysis relying on types. Type-based approaches have been applied to static analysis for security purposes. In contrast, TypeCopilot restores more type semantics from high-level language information, extending its scope to general pointers. Tools like RAP, MCFI, DR.CHECKER, IFCC, MLTA, TyPM [30, 31, 33, 37, 46, 47] resolve indirect-call targets with type matching, while approaches like TAT, TDI, DOPE and CAMP enhance data-flow integrity by allocating variables in separate memory regions according to their types [25, 32, 35, 48]. TypeCopilot is orthogonal and complementary to these works, with a focus on providing more comprehensive type information.

Type Inference. Prior works focus on inferring types for various scenarios, such as generating LLVM IR with inferred types[4, 8, 8, 11, 11, 15, 20, 39, 54, 58], and inferring types for high-level languages [34, 41, 49]. These methods cannot be directly applied to LLVM IR to resolve opaque pointers, as type inference must be co-designed with language-specific type organizations. Besides, most of them do not support multiple types, but our work does. TACAI [40] is a refinable IR for JAVA programs, which can also support multiple precise types. This work leverages the type inheritances and type-safe features of object-oriented languages. Our work focuses on low-level LLVM IR compiled from type-unsafe C language and particularly resolves generic pointers.

Points-to Analysis. The Andersen-style analysis [1, 45] traces the data flow to compute points-to sets, which is precise but computationally expensive. Steensgaard-style analysis [43] is also data-flow-based, but it sacrifices precision to improve scalability by design. Another scalable method is type-based, k-CFA [41], which was originally designed for higher-order languages. Our work leverages the type-based points-to analysis while introducing type-based access paths specifically designed to resolve generic pointers in LLVM IR.

7 Conclusion

We present TypeCopilot, a framework that infers accurate types for LLVM IR with opaque pointers through a multiple-type design. It restores type information from high-level languages and employs *TypeInfer* to handle complex IR instruction propagation. TypeCopilot enables type analyses to work with opaque pointers while improving precision by resolving IR-source code inconsistencies. Our integration with MLTA demonstrates its practicality and effectiveness. We open-source TypeCopilot to support the security analysis community.

8 Data-Availability Statement

Our tool is available at https://github.com/ZJU-SEC/TypeCopilot [59] under the MIT license. We publish our dataset for result reproduction and welcome contributions to enhance the tool.

Acknowledgments

We would like to thank all reviewers for their valuable comments sincerely. We also thank Ganhao Chen and Yunhe Wang for their contributions in refining our tool and artifacts. This work is partially supported by the National Key R&D Program of China (2022YFB3103900).

References

[1] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. (1994).

[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:25. doi:10.4230/LIPIcs.ECOOP.2016.2

[3] 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)*. USENIX Association, 1629–1645. https://www.usenix.org/conference/usenixsecurity21/presentation/bai

[4] Gogul Balakrishnan and Thomas Reps. 2007. DIVINE: discovering variables in executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation* (Nice, France) *(VMCAI'07)*. Springer-Verlag, Berlin, Heidelberg, 1–28. doi:10.1007/978-3-540-69738-1_1

[5] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*. Springer, Association for Computing Machinery, New York, NY, USA, 84–104. doi:10.1007/978-3-662-53413-7_5

[6] Mohamad Barbar and Yulei Sui. 2021. Compacting points-to sets through object clustering. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 159 (Oct. 2021), 27 pages. doi:10.1145/3485547

[7] Mohamad Barbar and Yulei Sui. 2021. Hash Consed Points-To Sets. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 25–48. doi:10.1007/978-3-030-88806-0_2

[8] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4, Article 65 (May 2016), 35 pages. doi:10.1145/2896499

[9] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1383–1400. https://www.usenix.org/conference/usenixsecurity24/presentation/cai-yuandao

[10] CFI [n. d.]. Control Flow Integrity Design Documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. (Accessed on 04/28/2024).

[11] Ligeng Chen, Zhongling He, and Bing Mao. 2020. CATI: Context-Assisted Type Inference from Stripped Binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 88–98. doi:10.1109/DSN48063.2020.00028

[12] Yueqi Chen and Xinyu Xing. 2019. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1707–1722. doi:10.1145/3319535.3363212

[13] Xiao Cheng, Jiawei Ren, and Yulei Sui. 2024. Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi-Point Slicing. *Proc. ACM Softw. Eng.* 1, FSE, Article 23 (July 2024), 23 pages. doi:10.1145/3643749

[14] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1034–1045. doi:10.1145/3445814.3446695

[15] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 51–60. doi:10.1145/2491956.2462165

[16] Hal Finkel. [n. d.]. The Type Sanitizer: Free Yourself from -fno-strict-aliasing. 2017 LLVM Developers' Meeting. https://llvm.org/devmtg/2017-10/slides/Finkel-The%20Type%20Sanitizer.pdf (Accessed on 08/21/2024).

[17] Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 256 (Oct. 2023), 30 pages. doi:10.1145/3622832

[18] The LLVM Compiler Infrastructure. 2024. LLVM. https://llvm.org/ [Accessed: 2024-04-27].

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. doi:10.1109/CGO.2004.1281665

[20] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *The Network and Distributed System Security (NDSS) Symposium 2011*.

[21] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A Hybrid Alias Analysis and Its Application to Global Variable Protection in the Linux Kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4211–4228. https://www.usenix.org/conference/usenixsecurity23/presentation/li-

guoren
[22] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1052–1065. doi:10.1145/3385412.3386020
[23] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 177–194. https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand
[24] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1963–1976. doi:10.1145/3548606.3560585
[25] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. 2024. CAMP: Compiler and Allocator-based Heap Memory Protection. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4015–4032. https://www.usenix.org/conference/usenixsecurity24/presentation/lin-zhenpeng
[26] LLVM. 2024. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html [Accessed: 2024-01-15].
[27] LLVM. 2024. Migration Instructions. https://llvm.org/docs/OpaquePointers.html#migration-instructions [Accessed: 2024-04-02].
[28] LLVM. 2024. 'tbaa' Metadata. https://llvm.org/docs/LangRef.html#tbaa-metadata [Accessed: 2024-04-02].
[29] LLVMLangRef:online [n. d.]. LLVM Language Reference Manual — LLVM 19.0.0git documentation. https://llvm.org/docs/LangRef.html. (Accessed on 04/27/2024).
[30] Kangjie Lu. 2023. Practical Program Modularization with Type-Based Dependence Analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1256–1270. doi:10.1109/SP46215.2023.10179412
[31] 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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867–1881. doi:10.1145/3319535.3354244
[32] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. 2023. DOPE: DOmain Protection Enforcement with PKS. In *Proceedings of the 39th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '23)*. Association for Computing Machinery, New York, NY, USA, 662–676. doi:10.1145/3627106.3627113
[33] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1007–1024. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry
[34] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 305–315. doi:10.1145/1806596.1806631
[35] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. 2022. Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation. In *2022 IEEE Symposium on Security and Privacy (SP) (Proceedings - IEEE Symposium on Security and Privacy, May)*. Institute of Electrical and Electronics Engineers Inc., United States, 1049–1065. doi:10.1109/SP46214.2022.9833675
[36] Peng Ning. 2014. Samsung KNOX and Enterprise Mobile Security. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (Scottsdale, Arizona, USA) *(SPSM '14)*. Association for Computing Machinery, New York, NY, USA, 1. doi:10.1145/2666620.2666632
[37] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 577–587. doi:10.1145/2594291.2594295
[38] opaque [n. d.]. Opaque Pointers — LLVM 19.0.0 git documentation. https://llvm.org/docs/OpaquePointers.html. (Accessed on 01/08/2024).
[39] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 690–702. doi:10.1145/3468264.3468607
[40] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. 2020. TACAI: an intermediate representation based on abstract interpretation. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (London, UK) *(SOAP 2020)*. Association for Computing Machinery,

New York, NY, USA, 2–7. doi:10.1145/3394451.3397204

[41] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda.* Carnegie Mellon University.

[42] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[43] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. doi:10.1145/237721.237727

[44] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: value-flow-based precise code embedding. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 233 (Nov. 2020), 27 pages. doi:10.1145/3428301

[45] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[46] PaX Team. 2015. Rap: Rip rop. In *Hackers 2 Hackers Conference (H2HC)*.

[47] 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)*. USENIX Association, San Diego, CA, 941–955. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[48] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 17–27. doi:10.1145/3274694.3274705

[49] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 131–144. doi:10.1145/996841.996859

[50] Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu, and Kui Ren. 2022. RegVault: hardware assisted selective data randomization for operating system kernels. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 715–720. doi:10.1145/3489517.3530549

[51] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 327–337. doi:10.1145/3180155.3180178

[52] Yutian Yang, Jinjiang Tu, Wenbo Shen, Songbo Zhu, Rui Chang, and Yajin Zhou. 2024. kCPA: Towards Sensitive Pointer Full Life Cycle Authentication for OS Kernels. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2024), 3768–3784. doi:10.1109/TDSC.2023.3334268

[53] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 89–106. https://www.usenix.org/conference/usenixsecurity22/presentation/yoo

[54] Dongrui Zeng and Gang Tan. 2018. From Debugging-Information Based Binary-Level Type Inference to CFG Generation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA) *(CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 366–376. doi:10.1145/3176258.3176309

[55] Yizhuo Zhai, Zhiyun Qian, Chengyu Song, Manu Sridharan, Trent Jaeger, Paul Yu, and Srikanth V. Krishnamurthy. 2024. Don't Waste My Efforts: Pruning Redundant Sanitizer Checks by Developer-Implemented Type Checks. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1419–1434. https://www.usenix.org/conference/usenixsecurity24/presentation/zhai

[56] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. 2021. Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 811–824. doi:10.1145/3460120.3484798

[57] 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)*. USENIX Association, Santa Clara, CA, 1205–1220. https://www.usenix.org/conference/usenixsecurity19/presentation/zhang-tong

[58] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. 2024. Plankton: Reconciling Binary Code and Debug Information. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 912–928. doi:10.1145/3620665.3640382

[59] Jinmeng Zhou, Ziyue Pan, Wenbo Shen, Xingkai Wang, Kangjie Lu, and Zhiyun Qian. 2025. *Artifact Evaluation Instructions for Type-Alias Analysis: Enabling LLVM IR with Accurate Types*. doi:10.5281/zenodo.15182810

[60] Jinmeng Zhou, Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed Azab, Ruowen Wang, Peng Ning, and Kui Ren. 2023. Automatic Permission Check Analysis for Linux Kernel. *IEEE Transactions on Dependable and Secure Computing* 20, 3 (2023), 1849–1866. doi:10.1109/TDSC.2022.3165368

[61] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. doi:10.1016/j.jss.2020.110884

Received 2025-02-27; accepted 2025-03-31