Redefining Indirect Call Analysis with KallGraph

Guoren Li
University of California, Riverside
gli076@ucr.edu

Manu Sridharan
University of California, Riverside
manu@cs.ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

***Abstract*—Call graph construction is a crucial prerequisite for a wide range of static analysis applications. State-of-the-art methods minimize precise but expensive pointer tracking by falling back to so-called "type analysis" which scales well to large programs such as the Linux kernel. In this paper, we undertake an in-depth evaluation and analysis of type-based methods that reveal new insights into flaws due to their ad-hoc nature. First, we find that in a number of cases, the soundness claims of recent type-based methods do not hold, resulting in missing indirect call targets. Second, we find the analysis is overly conservative in multiple aspects, leading to a large number of false indirect call targets. Based on these insights, we make the observation that such type-based methods can be converted into a hybrid pointer analysis framework that unifies the traditional pointer tracking methods and type-based methods. Based on such a framework, we develop a practical indirect call analysis that addresses both soundness and precision limitations. Our results demonstrate a remarkable level of soundness and precision improvements. KallGraph simultaneously improves precision and soundness by pruning up to 90% of indirect call targets and eliminating hundreds to thousands of missed indirect calls. Finally, KallGraph is fully parallelizable and can complete the analysis of Linux kernels in times ranging from tens of minutes to a few hours.**

1. Introduction

Call graphs maintain the relationship between caller and callee functions. They are widely used in inter-procedural program analysis for a variety of purposes [1–15].

Constructing call graphs is challenging because of indirect calls, which, in C programs, occur through dereferencing a function pointer. Classic points-to analysis can identify indirect call targets by tracking how a function address flows to a dereferenced function pointer. However, this approach is unscalable for large programs such as the Linux kernel [16].

This leads to the development of type-based indirect call analysis, offering superior scalability with decent precision and soundness [17, 18]. The most straightforward method along this direction is called function signature analysis (FSA) [17], which matches indirect calls and address-taken functions that share the same signature, i.e., return type and parameter types. However, this method is highly imprecise when resolving an indirect call with a common signature. To refine FSA, multi-layer type-based analysis (MLTA) [18] was proposed to refine the analysis results based on "type contexts" of indirect calls and function address-taken sites, i.e., matching the struct types a function pointer from which a function pointer is retrieved with the structure types into which a function address is stored. This method requires computing partial data flows (to track pointer propagation) and allows many indirect call targets to be pruned successfully. More recently, TyPM [11], KELP [19], TFA [20], SMLTA [21] were proposed as improved successors of MLTA.

In this paper, we conduct an in-depth evaluation and analysis of the state-of-the-art type-based methods and have two major observations: (1) MLTA's soundness claim does not hold due to flaws at the design level, and its precision is still limited due to conservative design regarding type handling; (2) Most of these limitations/flaws are not resolved by its successors [11, 19–21], causing still imprecise and unsound results. To address this, we map the MLTA algorithm that does partly data-flow and type-based analysis into a recent hybrid pointer analysis framework that unifies the two analyses. Through the framework, we develop a sound, precise, and scalable indirect call analysis, dubbed KallGraph, that revamps the current type-based analysis in a principled and systematic manner. Through extensive evaluation, we show that KallGraph outperforms state-of-the-art methods in both soundness and precision: (1) correcting thousands of false negative indirect call results of MLTA and its successors in large programs such as Linux kernels (with a subset of them verified by dynamic execution), (2) eliminating 75% to 90% of indirect call targets compared to MLTA, and over 60% compared to MLTA's successors. Finally, KallGraph is highly parallelizable, making it scalable for practical uses, i.e., tens of minutes to a few hours in our experiments.

In summary, we make the following contributions:

- An in-depth study revealing fundamental and previously unknown limitations in state-of-the-art type-based indirect call analysis, regarding both soundness and precision.
- An insight into the connection between existing type-based methods and a grounded hybrid pointer analysis, effectively addresses both soundness and precision limitations.
- A fully working and open-sourced indirect call analysis,

called KallGraph[1], based on the hybrid pointer analysis. The solution works well on large-scale programs such as the Linux kernel.

- A comprehensive evaluation demonstrating how KallGraph outperforms the existing methods with detailed data and case studies, which establishes a new standard.

2. Indirect Call Analysis

At a high level, there are two categories of approaches to resolve indirect calls (*icall* in short), points-to analysis and type-based methods. The points-to analysis treats the indirect call analysis as resolving the points-to set of the function pointer operand of the icall instruction. It keeps track of the object allocations and pointer propagation and resolves points-to set for pointers recursively. This makes it more precise but also resource-intensive. For example, two classic pointer analyses, Andersen's [22] and Steensgaard's [23] are both implemented in SVF [24, 25]. According to our experiments, when analyzing the Linux kernel, the former easily times out, while the latter spends 2 days generating an over-approximated result (§7.1). Other recent points-to analyses [26, 27] also fail to generate call graphs in days for large-scale programs such as the Linux kernel.

In contrast, instead of tracking the flow from address-taken functions to indirect calls, type-based methods bypass such tracking by conservatively assuming the function pointers "may" point to a target function simply because their types "match." Here we introduce a few type-based methods:

Function Signature Analysis (FSA). Traditionally, FSA [17] is widely used to resolve icalls for many applications such as CFI [1, 14, 28–30]. Given an icall, FSA considers all address-taken functions with the same signature (i.e., return type and parameter types) as targets. For example, Figure 1 shows `icall` at line 27 with type `rwptr`. FSA considers `m1_read()`, `m1_write()` and `m2_read()` as potential targets since they are address-taken and with the same type. This method is highly scalable because it is linear — only needs to check signatures across icall instructions and address-taken functions. It is also sound when type information is available and correct [31], including the source and destination types in type casts. However, it suffers high imprecision due to the over-approximation where false icall targets are included, e.g., `m1_read()` and `m2_read()`.

Multi-Layer Type Analysis (MLTA) [18] has become the de facto standard of indirect call analysis for many applications [2, 9, 12, 13, 32–36] targeting large-scale programs. It refines the FSA call graph through "type contexts" on top of the function signature. For the same example in Figure 1, MLTA will look at how the `icall`'s function pointer is retrieved from struct fields and which function addresses are stored in the same fields. For example, MLTA sees that `icall` is retrieved from `ops1->write`. This means that `m1_read()` and `m2_read()` are no longer possible icall targets because they are never stored in the same struct

1. https://github.com/seclab-ucr/KallGraph

```
1  typedef int (*rwptr)(char*);
2  void* gops = NULL;
3
4  // module1.h
5  struct M1 { rwptr read; rwptr write; } m1_ops;
6  int m1_read(char* str){ ... }
7  int m1_write(char* str){ ... }
8  void m1_init(){
9      m1_ops.read = m1_read; // struct M1, fd1
10     m1_ops.write = m1_write; // struct M1, fd2
11     gops = (void*) &m1_ops;
12 }
13
14 // module2.h
15 struct M2 { rwptr read; } m2_ops;
16 int m2_read(char* str){ ... }
17 void m2_init(){
18     m2_ops.read = m2_read; // struct M2, fd1
19 }
20
21 // module3.c
22 #include "module1.h"
23 struct M1* getops(){ return gops; }
24 int m3_exec(){
25     struct M1* ops = getops();
26     rwptr icall = ops->write;
27     return icall("/file"); // m1_write()
28 }
```

Figure 1. Code Snippet of Sample Indirect Calls

layer, denoted as {`struct M1, 2`} (2nd field of `struct M1`). Such refinement precision can be improved if there are additional "struct layers", hence the name of "multi-layer type analysis" (MLTA). For instance, a three-layer type context of `icall` may look like "`a->b->icall`", potentially allowing further refinement (by `a`) of targets.

However, it is not always possible to perform such refinements in MLTA. If we look at the type `struct M1` again which has an important statement in line 11, we will see a "type escape" where `struct M1*` is cast to `void*`. Unfortunately, `void*` is an unsupported type in MLTA, forcing it to fall back to FSA conservatively. This means that `icall` is still considered to call `m1_read()`, `m1_write()`, and `m2_read()`, as shown in Figure 2. MLTA reasons that `void*` pointers may subsequently be used in arbitrary ways, e.g., to be cast into many other types (more details in §3).

More recently, TyPM [11], KELP [19], TFA [20], and SMLTA [21] have been proposed to improve MLTA.

TyPM uses an additional dependence analysis to refine which types (e.g., struct layers in MLTA) could be accessed in which modules (i.e., source files). As shown in Figure 1, TyPM will figure out that there is no dependence relationship between `module2.h` and `module3.c`. Thus `struct M2` defined in `module2.h` will not be accessed in `module3.c`, and `m2_read()` will not be an icall target of `icall`. But TyPM cannot remove `m1_write()` from the MLTA result, as it is defined in `module1.h`, which was indeed dependent by `module3.c`. With such an additional refinement based on the module dependence analysis, TyPM is shown to eliminate 40% indirect call targets of MLTA for the Linux kernel.

KELP refines MLTA by dividing icalls into two categories: simple and complex. A simple icall means its target functions are propagated through short and simple paths from address-taken sites and can be handled through a "regional" (i.e., not whole-program) points-to analysis. A complex icall means the propagation is more complex to track (e.g., through many higher struct layers). For complex icalls, KELP falls back to MLTA. Similar to TyPM, KELP is shown to eliminate about 40% indirect call targets of MLTA. This is likely due to the fact that only 35% of icalls are simple.

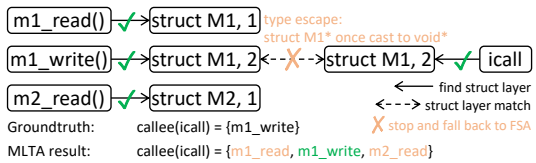

Figure 2. Type Escape of MLTA

TFA enhances MLTA by incorporating an inter-procedural, flow- and field-insensitive pointer analysis. Although TFA and KELP appear similar, their purposes diverge. KELP applies pointer analysis to a subset of "simple" indirect calls end-to-end, while TFA leverages pointer analysis to enhance type analysis performance, allowing more struct layers to be available. As a result, TFA claims to eliminate about 50% of the indirect call targets produced by MLTA.

SMLTA addresses several implementation-level issues in MLTA. While MLTA's design intends to refine indirect calls layer by layer (referred to as "multi-layer"), its implementation lacks precise maintenance of these layer relationships, causing potential precision issues. SMLTA identifies this problem and introduces a "strong multi-layer" approach, ensuring strict dependency between struct layers from lower to higher layers. As a result, SMLTA reduces MLTA's indirect call targets by approximately 30%.

3. In-depth Analysis of SOTA Methods

In this section, we comprehensively analyze and evaluate the design of these state-of-the-art (SOTA) methods, identifying fundamental and previously unknown limitations that prevent them from being more precise and sound. Since TyPM, KELP, TFA, and SMLTA are recent successors of MLTA, we focus more on MLTA, as most of its imprecision and unsoundness are also inherited by its successors.

MLTA refines icall targets based on multi-layer struct type contexts. Its algorithm contains three components and is supported by three corresponding type refinement rules [18]:
(1) *target resolving rule*: collect higher struct layers for icalls, i.e., from which structs the function pointers are retrieved.
(2) *type confinement rule*: collect higher struct layers for function addresses, i.e., which structs they get stored into.
(3) *type propagation rule*: match the collected struct layers from both ends. Roughly speaking, if the struct layers of the two ends are compatible, the function is considered a target.

Correspondingly, there are two important requirements in order to make this idea work well:
(1) MLTA successfully and comprehensively finds the struct layers around both address-taken function end and icall end.
(2) The identified struct layers are supported for refinement.

Through an in-depth analysis, unfortunately, we identify several fundamental limitations in both precision and soundness of MLTA (and its successors). For example, the first requirement is often not met due to two reasons:

① **Imprecision in the Target Resolving Rule** — Conservative fallback behavior when struct layers for icalls are missing. To retain scalability, MLTA chooses to find higher

```
1  int a_read(char*){ ... }
2  int b_read(char*){ ... }
3  int c_read(char*){ ... }
4  int f_read(char*){ ... }
5
6  typedef int (*rwptr)(char*);
7  struct A { rwptr read = a_read; } aops;
8  struct B { rwptr read; } bops;
9  struct C { rwptr read = c_read; } cops;
10 struct E { rwptr read; };
11 struct F { rwptr read; };
12
13 rwptr intra(rwptr arg){
14     rwptr icall1 = arg;
15     icall1("/file"); // a_read()
16     return b_read; // &b_read --> bops.read
17 }
18 void unsound_confinement(){
19     // {struct A, 1} --> arg
20     bops.read = intra(aops.read);
21 }
22 void call_from_b(struct B* b){
23     rwptr icall2 = b->read;
24     icall2("/file"); // b_read()
25 }
26 void imprecise_cast(){
27     rwptr icall3 = aops.read;
28     icall3("/file"); // a_read()
29 }
30 void c_to_a(struct C* c){ // C cast to A
31     struct A* a = (struct A*) c;
32     rwptr icall4 = a->read; // {struct A, 1}
33     icall4("/file"); // a_read() or c_read()
34 }
35 void unsound_cast(struct E* e){ // E cast to F
36     struct F* f = (struct F*) e;
37     f->read = f_read; // f_read() stores to F
38     rwptr icall5 = e->read; // use by E
39     icall5("/file"); // f_read()
40 }
```

Figure 3. An Example Illustrating Limitations of MLTA

struct layers *intra-procedurally*. However, if no struct layers are found for a given icall, MLTA will fall back to FSA, even though the struct layers may be available if searched inter-procedurally. As shown in Figure 3, intra-procedural analysis cannot figure out `icall1` comes from `aops.read` (i.e., the first field of `struct A`, or {`struct A`, `1`} in short) in line 20. This means that MLTA will fall back to FSA for `icall1`, resulting in false icall targets such as `b_read()`, `c_read()`, and `f_read()`. Only KELP and TFA realized this limitation and tried to address it by integrating inter-procedural pointer analysis. However, since the applied pointer analyses are limited in scope, they can still fall back to FSA (see "complex icall" in KELP [19] and "fall back strategies" in TFA [20]). This means they can only partially resolve this limitation, leaving the imprecision still (§7.1).

② **Unsoundness in the Type Confinement Rule** — Incorrect handling of address-taken functions with missing struct layers. Unlike the fallback behavior when MLTA fails to find structs for icalls, MLTA's type confinement rule does not consider fallback when it fails to find structs for address-taken functions — leading to a problematic asymmetry. MLTA omits such functions from being targets of any icalls, as long as struct layers are found for those icalls. For example, we see that `b_read()` is returned from `intra()` in line 16 to `unsound_confinement()` in line 20, and subsequently stored to `bops.read`. However, due to the intra-procedural limitation, MLTA fails to find this struct layer. It erroneously decides it should not be `icall2`'s target because `b_read()` does not have a layer and `icall2` does. Interestingly, none of the MLTA successors realizes such a significant asymmetry. As a result, according to our experiment on open-sourced successors (i.e., TyPM and SMLTA), they inherited all the related unsound cases (missed icall targets) from MLTA.

Next, even if we assume MLTA finds the higher struct layers successfully and completely on both ends, the second requirement is also often not met due to two reasons:

③ **Imprecision in the Type Propagation Rule** — Conservative handling of type cast and type escape. MLTA conservatively handles type casts by maintaining global cast relationships at the type level, in a type propagation map. The associated type propagation rule has two kinds of imprecision. First, as mentioned in §2, when MLTA considers a destination type to be unsupported, e.g., `void*`, it simply considers the source type escaped and ineligible for type refinement; this effectively leads to a fallback to FSA. Second, even if the type is supported, it can still lead to imprecision. Considering `icall3` in Figure 3, MLTA will find it is from {`struct A, 1`} (line 27). Because of the cast instruction from `struct C` to `struct A` in line 31, MLTA conservatively considers all `struct A` instances to be aliases of `struct C` instances. Thus, not only `a_read()` but also `c_read()` are considered targets of `icall3`. However, `c_read()` is clearly a false target because the cast instruction does not impact `icall3` in line 27. `icall4`, on the other hand, is indeed affected by the cast instruction, and therefore it is correct to consider both `a_read()` and `c_read()` as targets. All of the MLTA successors inherited this imprecision since they handled the type cast similarly at the type level.

④ **Unsoundness in the Type Propagation Rule** — Inclusion-based handling of type casts. To handle type cast, MLTA maintains a global type propagation map that collects cast relationships between types in an inclusion manner [18], i.e., a directional relationship. Specifically, given a type cast instruction, the type propagation rule considers only destination type instances to be aliases of the source type instances. However, the converse should also be true; otherwise, it is not sound. Let us focus on `icall5` in Figure 3. MLTA identifies that `icall5` originates from {`struct E, 1`}. Unfortunately, despite the type cast from `struct E*` to `struct F*`, MLTA does not treat `e` as an alias of `f` by its inclusion rule, while only considers the opposite to be true. This means that `f_read()` will be incorrectly missed as a potential icall target of `icall5`. This unsound design persists in all MLTA successors.

⑤ **General Limitations and Flaws —** In addition to the abovementioned design-level limitations, after our empirical experiments and careful analysis, we find SOTA methods introduce more false negatives in several ways: (1) TyPM, KELP, and TFA depend on an initial call graph generated by MLTA, establishing an unreliable foundational premise. (2) Their implementation underestimates the complexity of raw LLVM IR. They usually overlook complex instructions containing embedded operators, such as `BitCastOperator`. Additionally, certain instructions are mishandled; for instance, `Phi` and `Select` instructions are treated with "must" logic instead of "may", leading to soundness issues. (3) There is a tendency to aggressively use highly optimized LLVM IRs of target programs, neglecting common pitfalls associated with these "broken" IRs (e.g., type information eliminated by compiler optimizations). While this approach may improve precision and scalability on optimized IRs, it compromises soundness, rendering the results unreliable for applications that require sound guarantees, such as control flow integrity.

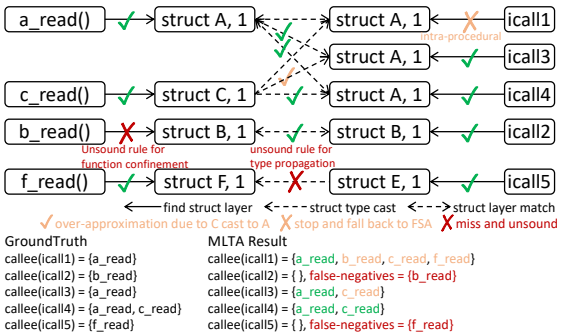

Figure 4. Unsound and Imprecise MLTA.

4. Problem Formulation and Insights

We observe the limitations of the type-based methods fundamentally stem from the coarse-grained and unsound tracking and matching of types. We reformulate the methodology that MLTA aims to develop but has not achieved.

The basic intuition we distill behind MLTA is *once an address-taken function flows into a struct field, the program must access the same type of struct field(s) to retrieve the value (e.g., a function pointer) before using it to make an indirect call.* In particular, we observe that MLTA attempts to perform some limited data-flow analysis (i.e., traditional points-to in an intra-procedural fashion) as well as the type-based methods. Based on this intuition, we outline three steps that are needed to support a working algorithm.

① Given a function address, compute the struct fields it may propagate into via a thorough on-demand points-to analysis.
② Find the aliases of the same or compatible struct fields.
③ Trace those aliases and continue the points-to analysis to see whether the struct fields will be eventually evaluated to indirect calls, i.e., function pointer dereference.

While this process is conceptually reasonable, its realization matters significantly to the outcome in terms of precision, scalability, and soundness. Let us revisit the example in Figure 3. To aid our understanding, we visualize the program in a data-flow graph shown in Figure 4. In the graph, we illustrate how the three steps are carried out. On the left, we have four address-taken functions and on the right, we have five icalls. In between, there are struct layers (only one layer in this example). The solid lines represent the on-demand points-to analysis that should be carried out in the first and third steps. The dotted lines represent the type refinement, in the second step, to match aliases based on struct layers. Finally, we list the ground truth and MLTA results for each icall at the bottom.

For `a_read()`, it first propagates into a node of type {`struct A, 1`}, by considering all the nodes of type {`struct A, 1`} aliases, through them, `icall3` and `icall4` can be reached. However, `a_read()` is unable to reach `icall1`. Because the MLTA's intra-procedural analysis will

fail to link {struct A, 1} with icall1. More interestingly, MLTA "accidentally" infers a_read() to be a target of icall1 because it falls back to FSA regarding icall1, which introduces three false targets at the same time.

For b_read(), it is a false target of icall1 again due to the fallback of icall1. It should have been considered a target of icall2, but we see a missing link from {struct B, 1} to icall2 due to MLTA's unsound type confinement.

c_read() is correctly recognized as a target of icall4. However, MLTA also falsely reports that c_read() is a target of icall1 and icall3. The former is because of the fallback again. And the latter is due to the imprecise type propagation rule where a false link from {struct C,1} to {struct A,1} is created.

f_read() is missed by MLTA as a target of icall5, since the unsound inclusion-based type propagation rule will miss the link from {struct F,1} to {struct E,1}.

The above analysis motivates us to seek a more principled solution for the process outlined here, specifically by adhering to key design principles in the execution of the three steps:

(1) The on-demand points-to analysis should exhaustively find available struct layers (instead of only intra-procedurally). This affects ① and ② issues we pointed out in §3.

(2) When finding the aliases of the struct layers, we should not view the relationship at the type-level (e.g., type propagation rule in MLTA), particularly in the presence of type cast. Instead, we should consider object-level relationships and avoid overly conservative and imprecise heuristics, such as type escape. This pertains to ③ and ④ in §3.

(3) The points-to analysis should follow a sound implementation (e.g., Andersen-style analysis). For example, it should record all struct layers that an address-taken function propagates into, which is not implemented correctly in MLTA – this relates to ⑤ in §3.

In summary, we show how the type-based analysis can be augmented systematically by on-demand points-to analysis. Note that the end-to-end solution we sketched in this section is inherently individualized: each address-taken function will be analyzed separately. Even though such fine-grained and precise tracking can be costly, the scalability concern is mitigated by the fact that the solution is highly parallelizable.

5. Design

To realize the principles outlined in §4, we develop KallGraph, which aims for a sound, precise, and scalable indirect call analysis that outperforms SOTA methods. In this section, we present the design of KallGraph.

To incorporate points-to analysis into the type-based analysis in a proper way, we first examine prior works on hybrid pointer analysis [13, 37–39], which improve points-to analysis scalability by selectively applying approximation strategies, formulated via the context-free-language reachability (CFL-reachability) framework [40]. Then we map and shape the hybrid idea to bridge the gap between points-to-based and type-based indirect call analysis. More specifically, we build KallGraph based on one of the most recent hybrid pointer analysis frameworks, Unias [13] (§5.1), making the connection between hybrid analysis and MLTA (§5.2), and contributing a few important improvements (§5.3, §5.4, §5.5).

5.1. CFL-Reachability and Unias

CFL-Reachability Analysis. In this technique, a program is represented as a graph where variables are nodes and instructions operating on those variables are edges. CFL-reachability analysis then transforms the program analysis problem into a context-free language (CFL) graph reachability problem [40].

Consider the following program as an example:

```
1  struct A{
2      int* fd0;
3      int* fd1;
4  }a;
5  int obj;
6  int* ptr;
7
8  void func1(){
9      a.fd1 = &obj;
10 }
11 void func2(){
12     ptr = a.fd1;
13 }
```

This program is represented as a graph:

$$obj_{addr} \xrightarrow{Store} a.fd1_1 \xleftarrow{Gep_{A,1}} a \xrightarrow{Gep_{A,1}} a.fd1_2 \xrightarrow{Load} ptr$$

In the graph, we can see the address of variable obj is stored in the second field[2] (i.e., fd1) of variable a, while the same field is loaded to variable ptr elsewhere. In other words, ptr might point to obj. To realize such a points-to relationship by CFL-Reachability analysis, we can follow field-sensitive Andersen's on-demand points-to analysis rules, which we list a part relevant to the example below:

$$\mathbf{F} \rightarrow (\ Store\ \mathbf{I\text{-}Alias}\ Load\ )*$$
$$\mathbf{I\text{-}Alias} \rightarrow (\ \overline{Gep_{t,i}}\ \mathbf{I\text{-}Alias}\ Gep_{t,i} \mid \varepsilon\ )*$$

Here the $Edge$ and $\overline{Edge}$ represent forward and backward edges respectively.

To find pointer aliases of obj (i.e., pointers that point to obj), we can start from its address node (i.e., obj_{addr}), and parse the production **F** to reach the pointer alias nodes (i.e., obj_{addr} **F** $ptr \rightarrow obj_{addr}\ Store\ \overline{Gep_{A,1}}\ Gep_{A,1}\ Load\ ptr$). By applying the above two rules, we can easily determine that the path conforms to the language generated by the grammar. In other words, this indicates that node ptr can be reached from node obj_{addr}, i.e., ptr might point to obj.

Unias is a hybrid alias analysis framework [13]. It introduces the notion of type-based shortcut edges, allowing the graph traversal (during the CFL reachability analysis) to skip large portions of the graph, trading off precision for scalability. Instead of relying on the unsound heuristics in MLTA which can be viewed as effectively creating shortcuts from the graph traversal perspective, Unias guarantees that traversing added shortcut edges (instead of the original paths) preserves soundness compared to a standard Andersen-style analysis. In particular, Unias incorporated new type-based CFL reachability rules that are formally reasoned, handling critical issues such as pointer type casting more precisely than prior type-based reasoning.

2. A $Gep_{t,i}$ instruction represents the field-addressing, i.e., computing an offset of the *i-th* field relative to the base of type *t*. GEP is taken from the LLVM terminology [41].

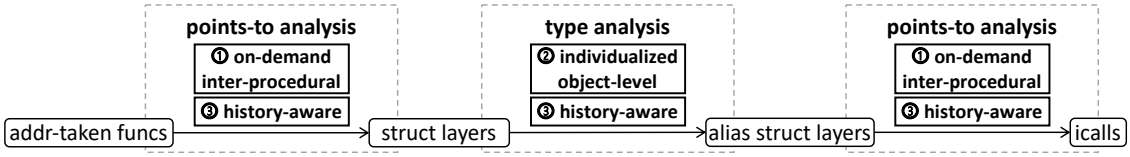

Figure 5. High Level Workflow of KallGraph

Taking the `icall2` and `b_read()` example in Figure 3:

$$b\_read_{addr} \xrightarrow{Store} bops.read \xleftarrow{Gep_{B,0}} bops \rightarrow ... \rightarrow b \xrightarrow{Gep_{B,0}} b.read \xrightarrow{Load} icall2$$

Suppose the path from `bops` to `b` is long and overly expensive to traverse by Andersen's points-to rules. Unias can transform the graph by inserting a type-based shortcut edge to avoid dataflow tracking. In this example, the graph will be transformed into the following:

$$b\_read_{addr} \xrightarrow{Store} bops.read \xrightarrow{Shortcut_f} b.read \xrightarrow{Load} icall2$$

Here the $Shortcut_f$ represents a shortcut introduced by directly matching the pair of $Gep_{B,0}$ edges.

By adding a rule from Unias:

$$\textbf{I-Alias} \rightarrow Shortcut_f$$

We can easily derive that `icall2` might call `b_read()`.

It is important to note that, starting from the address node of `b_read()`, Unias will only reach `icall2`, since there is no reachable path to any other icalls in Figure 3. In contrast, as discussed in §3, MLTA falsely identifies `b_read()` as a target of `icall1` by falling back to FSA.

Unias can also successfully and precisely resolve the type cast to `void*` (i.e., type escape) example in Figure 1:

$$m1\_write_{addr} \xrightarrow{Store} m1\_ops.write \xleftarrow{Gep_{M_1,1}} m1\_ops \xrightarrow{Assign_1} gops \xrightarrow{Assign_2} ops \xrightarrow{Gep_{M_1,1}} ops.write \xrightarrow{Load} icall$$

Here $Assign_1$ represents the cast instruction at line 11 (i.e., assign the value from `&m1_ops` to `gops`), while $Assign_2$ represents the call instruction at line 25 (i.e., assign the value from `gops` to `ops`).

The graph will be transformed into the following:

$$m1\_write_{addr} \xrightarrow{Store} m1\_ops.write \xrightarrow{Shortcut_{c-M_1,1}} gops \xrightarrow{Assign_2} ops \xrightarrow{Gep_{M_1,1}} ops.write \xrightarrow{Load} icall$$

Here the $Shortcut_{c-M_1,1}$ represents a shortcut introduced by a `struct M1` cast instruction (i.e., $Assign_1$).

By adding a rule from Unias:

$$\textbf{I-Alias} \rightarrow Shortcut_{c-t,i}\ \textbf{I-Alias}\ Gep_{t,i}$$

We can infer that `icall` might call `m1_write`. Note that the $Shortcut_{c-M_1,1}$ started at a field node $m1\_ops.write$, and ended at an object (i.e., non-field) node $gops$. This means that Unias will continue searching for a field node from $gops$ (i.e., it will find $ops.write$ after traversing the $Gep_{M_1,1}$ edge). We wish to point out that starting from address nodes of `m1_read` and `m2_read` will not reach node $icall$.

Similarly, for the type cast from `struct C` to `struct A` example in Figure 3, Unias can precisely derive that `c_read` will only be a target of `icall4` (i.e., excludes the false `icall3` in MLTA). Since the $Shortcut_{c-t,i}$ edge will only be introduced from node $cops.read$ to node a (i.e., by cast instruction at line 31), and only $icall4$ is reachable after a. Such an object-level cast handling is superior to SOTA type-based methods.

5.2. From MLTA to Hybrid Pointer Analysis

We are the first to make the observation that the analysis performed by MLTA is essentially an ad-hoc version of the hybrid pointer analysis where it conducts a combination of data flow and type-based analysis. As shown in Figure 5, we map the hybrid pointer analysis in Unias to the following three components leveraged by KallGraph which overcomes the deficiencies of the ad-hoc hybrid analysis in MLTA:

① ***On-demand inter-procedural points-to analysis*** performs exhaustive inter-procedural searches before and after struct layers (used in both the 1st and 3rd steps). More specifically, KallGraph follows the on-demand version of Andersen's points-to analysis rules, which enable precise points-to analysis while preserving soundness [22]. We use field-sensitivity for precision, and inter-procedural analysis for an exhaustive and complete search. We use flow-insensitivity due to the multi-entry nature [13] of large-scale programs such as Linux kernels, and context-insensitivity for scalability.

② ***Individualized type analysis*** that enables precise type alias matching at the object-level (used in the second step). By pre-processing all the cast instructions, KallGraph will first store each pair of cast nodes (rather than their types) into a CastMap, thus considering the cast relationship at object-level (instead of type-level) [13]. Recalling the `c_read()` example discussed in §5.1, the state-of-the-art type-based reasoning [18] assumes all instances of `struct A` are aliased with `struct C`, leading to an over-approximation. However, we only need to consider the specific instance of `Struct A* a` at line 31 aliased with `Struct C*`, because it is the only pointer that has experienced cast related to `Struct C`.

③ ***History-aware*** is a property provided by the nature of CFL-reachability that enables KallGraph to precisely track the propagation of the address-taken function before and after the alias struct layers — it remembers the states accumulated during the on-demand points-to analysis before taking the type-based shortcuts, and will continue to match edges afterward in the subsequent on-demand points-to analysis.

With these components, given the two examples in Figure 2 and Figure 4, KallGraph handles them precisely and

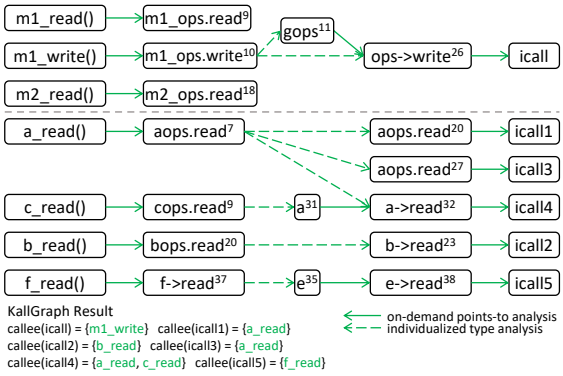

Figure 6. Sound and Precise KallGraph.

soundly, as shown in Figure 6. Note that we have changed those "struct layers" to concrete variables (with their line numbers as $\texttt{var}^{\texttt{line}}$) in the figure, showing the substantial difference between KallGraph's individualized type analysis and SOTA methods' type-level type analyses.

In the rest of this section, we will present the important improvements of KallGraph beyond Unias.

5.3. Comprehensive CFL-Reachability Rules

Unias claims to handle various pointer arithmetic by relaxing rules on GEP instructions, which unfortunately still suffers from real-world false negatives.

The reason is that Unias assumes the GEP instructions and offsets should always be matched as pairs, i.e., `(X, -X)`. However, we find two common types of counterexamples in the Linux kernel: `(X, [nothing])` — a GEP without a matching counterpart, and `(X+Y, -Z)` where `Z == X+Y`.

The first case is introduced because LLVM sometimes uses $Gep_{t,0}$ (i.e., zero offset) as literally $Assign$. Such a singleton GEP instruction never needs to be paired with a counterpart. The second case is commonly seen where two normal field accesses $Gep_{t_1,X}$ and $Gep_{t_2,Y}$ with a negative arithmetic $Gep_{t_3,\text{-}Z}$ that matches the first two GEPs.

To address this, KallGraph redesigns the CFL-Reachability rules related to Gep edges to enable richer arithmetic reasoning. Instead of strictly constraining Gep edges to be paired, KallGraph allows arbitrary Gep edges to be traversed during the analysis. Meanwhile, KallGraph maintains an offset stack to track cumulative Gep offsets, and only checks if the offset is zero at the point of $Load$ or $\overline{Store}$ edge. Such a re-design essentially considers all Gep edges as $Assign$ edges with offsets (in other words, $Assign$ edge is a Gep edge with offset zero), and will only check the cumulative offset when necessary. This design unifies the Gep and $Assign$ edges and will reduce the false negatives of Unias in the real world.

As a result, for rules 1-12 of Unias (§A), KallGraph unifies them into the following form:

$$\mathbf{F} \rightarrow (\ Assign \mid Gep_{\ t,o} \mid \overline{Gep_{\ t,\text{-}o}} \mid Store\ \textbf{I-Alias}\ Load\ )*$$
$$\overline{\mathbf{F}} \rightarrow (\ \overline{Assign} \mid Gep_{\ t,o} \mid \overline{Gep_{\ t,\text{-}o}} \mid \overline{Load}\ \textbf{I-Alias}\ \overline{Store}\ )*$$
$$\textbf{I-Alias} \rightarrow \overline{\mathbf{F}}\ \textbf{I-Alias}\ \mathbf{F} \mid \overline{Load}\ \textbf{I-Alias}\ Load \mid \varepsilon$$

Algorithm 1: Algorithm of KallGraph

Input : *cur* (current node, initially the address of a function)
Output : *iCalls* (result indirect calls)

```
1  MHS ← ∅; iCalls ← ∅;
2  GoForiCall(cur, state):
3      DependenceChecks(cur, state)
4      if isiCall(cur) && isEmpty(MHS)
5          iCalls.insert(cur)
6      for each cur --Assign--> nxt
7          GoForiCall(nxt, S_f)
8      for each cur --Store--> nxt
9          MHS.push(0)
10         GoForiCall(nxt, S_b)
11         MHS.pop()
12     if !isEmpty(MHS)
13         offset = MHS.top
14         for each cur --Load--> nxt
15             if offset == 0
16                 MHS.pop()
17                 GoForiCall(nxt, S_f)
18                 MHS.push(0)
19         for each cur --GEP_{t,i}--> field
20             offset -= byte({t, i})
21             GoForiCall(field, S_f)
22             offset += byte({t, i})
23         if state == S_b
24             for each cur <--Assign-- nxt
25                 GoForiCall(nxt, S_b)
26             for each cur <--Load-- nxt
27                 MHS.push(0)
28                 GoForiCall(nxt, S_b)
29                 MHS.pop()
30             for each cur <--Store-- nxt
31                 if offset == 0
32                     MHS.pop()
33                     GoForiCall(nxt, S_b)
34                     MHS.push(0)
35             for each cur <--GEP_{t,i}-- base
36                 offset += byte({t, i})
37                 GoForiCall(nxt, S_b)
38                 offset -= byte({t, i})
39             for each cur --Shortcut_f--> field
40                 GoForiCall(field, S_f)
41             for each cur --Shortcut_{c-t,i}--> nxt
42                 offset += byte({t, i})
43                 GoForiCall(nxt, S_b)
44                 offset -= byte({t, i})
```

Taking an example, we might see dataflow as follows:

$$a \xrightarrow{Store} b \xleftarrow{Gep_x} c \xleftarrow{Gep_y} d \xleftarrow{Gep_z} e \xrightarrow{Load} f$$

Here node f should be reached from node a if `x + y == z`. But Unias assumes Gep edges are paired and will stop at node e. Differently, KallGraph will only check if the cumulative offset is zero when encountering the $Load$ edge, thus reaching node f successfully.

At a high level, KallGraph (1) consolidates the four fundamental edges, $Store$/$Load$/$Assign$/Gep into three, (2) only requires pairing $Store$ and $Load$ edges (i.e., before and after **I-Alias**), reflecting the paired nature of memory reference and dereference. These two principals not only

contribute valuable insights to CFL-reachability rule design but also unleash better real-world performance.

After applying the new CFL-reachability rules, we present the algorithm of KallGraph in Algorithm 1.

In brief, finding the icalls for a given address-taken function `func` is equivalent to invoking `GoForiCall(`$func_{addr}$`,` S_f`)`, where the $func_{addr}$ represents the address node of `func` and S_f represents the analysis state is currently in the production **F** or the right part of the production **I-Alias**. Similarly, S_b represents the state of production $\overline{\mathbf{F}}$ (i.e., the reverse of **F)** or the left part of the production **I-Alias**.

As discussed before, we employ MHS (memory history stack) as the offset stack that tracks cumulative offsets for *Gep* edges. This stack essentially adds the additional offset check associated with CFL-reachability rules. Given that Algorithm 1 can be viewed as a depth-first-search (DFS) algorithm, at lines 15 and 31, the analysis will check if the stack top offset is zero, if so, the analysis proceeds along the current path because it finds a matching *Store-Load* pair. Otherwise, it stops exploring the current path.

Here the helper function `byte({t,i})` is used to evaluate the byte offset given the type and index of a $Gep_{t,i}$. We will introduce the `DependenceChecks()` (line 3) in §5.4.

5.4. Boostrapping via an Optimized Fixed-Point Algorithm

Similar to other static analysis work [8–11, 19, 20], Unias requires a preliminary call graph generated by MLTA as input, which results in soundness issues given the unsound nature of MLTA. Unlike Unias, KallGraph bootstraps from an empty call graph (i.e., no icalls are resolved initially). Hence, to handle interdependent icalls (e.g., in Figure 7, resolving `icall2` requires to resolve `icall1` first), KallGraph needs to iteratively conduct the indirect call analysis. Exhaustive points-to analyses can use on-the-fly call graph construction [42] to track dependent icalls, but to our best knowledge, all previous demand-driven analyses rely on a preliminary call graph. KallGraph is the first on-demand indirect call analysis with fully on-the-fly reasoning for call targets.

```
1  int a_read(char*){ ... }
2  int b_read(char*){ ... }
3  typedef int (*rwptr)(char*);
4  void bar(rwptr param){
5      rwptr icall2 = param;
6      icall2("/file"); // a_read()
7      rwptr ret = b_read;
8      return ret;
9  }
10 void foo(){
11     void (*icall1)(rwptr) = bar;
12     rwptr arg = a_read;
13     rwptr fptr = icall1(arg); // bar()
14     icall3 = fptr;
15     icall3("/file"); // b_read()
16 }
```

Figure 7. Dependent relationship between icalls

A straightforward but expensive way to achieve on-the-fly reasoning is to exhaustively analyze all address-taken functions iteration by iteration until a fixed point (no additional call graph edges are introduced). However, since KallGraph operates on an on-demand basis, i.e., analyzing each address-taken function individually, we use a more efficient approach.

Algorithm 2: Optimized Fixed-Point Algorithm

```
1  DependenceChecks(cur, state):
2      if isiCallArg(cur) && state == S_f
3          DepiCalls[func].insert(getUseiCall(cur))
4      if isFuncRetVar(cur) && state == S_f
5          DepFuncs[func].insert(getUseFunc(cur))
6      if isiCallRetVar(cur) && state == S_b
7          DepiCalls[func].insert(getUseiCall(cur))
8      if isFuncParam(cur) && state == S_b
9          DepFuncs[func].insert(getUseFunc(cur))
10 iterativeAnalysis(funcs):
11     for each func ∈ funcs
12         GoForiCall(func, S_f)
13     nextIterFuncs ← ∅
14     for each func ∈ funcs
15         for each icall ∈ DepiCalls[func]
16             if hasNewCallees(icall)[]
17                 nextIterFuncs.insert(func)
18         for each callee ∈ DepFuncs[func]
19             if hasNewCallers(callee)[]
20                 nextIterFuncs.insert(func)
21     if nextIterFuncs.size > 0
22         iterativeAnalysis(nextIterFuncs)
```

In the first iteration, we have to analyze all address-taken functions exhaustively. However, in subsequent iterations, we only need to re-analyze a subset of the address-taken functions in each iteration. Specifically, if an address-taken function node n reaches an icall node i during the reachability analysis, e.g., as a function argument, we will consider n to be "dependent" on i, and re-analyze the flow of n in the next iteration if i obtains a new target.

We use an example in Figure 7 to illustrate this. In the first iteration, by analyzing `a_read()`, we will observe it is dependent on `icall1` since it gets passed as an argument to `icall1`. At that point, we are not aware of `icall1`'s target (i.e., `bar()`). In the second iteration, `icall1` is now resolved to `bar()`, we will re-analyze `a_read()` and realize it is a target of `icall2`. The same idea goes to `b_read()`, which is dependent on function `bar` as a return value.

The algorithm of the optimized fixed-point is represented in Algorithm 2. We use `DependenceChecks()` (line 3 in Algorithm 1) to collect the dependent relationship by checking the properties of each traversed node (e.g., check if it is an icall argument node by `isiCallArg(node)`). As a result, for the example in Figure 7, we will collect two dependency maps as follows:

DepiCalls[`a_read`] = {`icall1`} **DepFuncs**[`b_read`] = {`bar`}

For the example in Figure 7, at the end of the second iteration, KallGraph will reach the fixed-point since no new dependency is introduced (i.e., *nextIterFuncs.size == 0*).

5.5. Optimized Minimal CastMap

As mentioned in §5.2, KallGraph utilizes a CastMap inherited from Unias that precisely handles type casting at the object-level. However, such a "precise" CastMap still suffers from precision and scalability issues. An important

observation we make is that many type casts only exist locally. For example, when a `struct A` object is copied by `memcpy(void* dst, void* src)`, its pointer will always be first cast to `void*`, then passed into `memcpy()`, later on, the new object pointer `dst` will be cast back to `struct A*`.

If we handle this with the CastMap, given the two casts (one from `struct A*` to `void*` and one from `void*` to `struct A*`), we will have to apply on-demand points-to analysis to explore the dataflow before/after the `src`/`dst` pointers whenever utilizing shortcuts provided by a `struct A`, which already sacrifices the notion of "shortcut" in Unias, making the analysis both less precise and less scalable. In fact, in most cases, the cast instructions are introduced "locally" so they can be safely ignored from CastMap if we can confirm there is no side effect.

To address this, when building the CastMap, instead of blindly collecting all cast relationships among nodes, KallGraph performs an additional "look-ahead" on-demand points-to analysis for each cast, and will only collect those non-trivial casts. And those trivial casts will never be stored in the CastMap which thus is built minimally.

6. Implementation

We implement KallGraph on top of Unias [13] which is based on LLVM and SVF [24]. KallGraph has 2.1K SLOC, with 1.8K being changed to Unias. Below we describe several important aspects of the implementation.

Program Representation. Like Unias, KallGraph leverages the SVF Pointer-Assignment-Graph (PAG) [43], a widely used directed graph representation of a program that is specifically designed for static analysis, constructed by processing the LLVM IR. The four fundamental edges *Store*/*Load*/*Assign*/*Gep* are the ones that PAG abstracts.

Consistent Memory Model. Similar to Unias, for a better real-world soundness performance, KallGraph uses byte offset instead of field index to unify *Gep* offsets among different struct types. However, the Unias byte offset model still relies on the SVF flattened index model, which also assumes *Gep* edges to be paired as mentioned in §5.3. To integrate our new *Gep* related rules, we implement an accurate byte offset model that evaluates the byte offset given a *Gep* edge based on its raw LLVM instruction.

Also, the SVF flattened index model faces some consistency issues when dealing with arrays and structs. For example, the offset evaluation of "`a->b[i]->c`" vs. "`a->b + b[j]->c`" should be the same which unfortunately is not true in SVF. KallGraph also takes care of this in the accurate byte offset model. As a result, KallGraph is strictly array-insensitive. For example, given an extremely nested array and struct access example like `a.b[i].c.d[j].e`, the evaluated offset remains consistent no matter whether `i` or `j` is a variable or constant number.

Anonymous Struct. There are many anonymous structs in LLVM IR, which are introduced by either the source code or the compiler. KallGraph models them in the form of {fieldNum, sizeInByte}, where the first element represents the field number of the anonymous struct, and the second element represents the struct size in byte. When leveraging shortcuts for a named struct, we will also look up the anonymous structs that share the same fieldNum and sizeInByte.

7. Evaluation

In this section, we conduct an extensive evaluation of KallGraph to assess its real-world effectiveness. We focus on the following three standard metrics: (1) precision (§7.1), (2) soundness (§7.2), and scalability (§7.3).

Experiment Setup. To evaluate KallGraph, we choose 6 large-scale C/C++ target programs where indirect calls are prevalent: Xen-4.18 (637K LoC), QEMU-8.1.4 (2.05M LoC), Wine-8.16 (6.13M LoC), Freebsd-14 (15.41M LoC), Linux-5.15 (20.83M LoC), and Linux-6.5 (24.09M LoC). We use LLVM-14.0.6 and SVF-2.5. These programs are compiled by Clang-14.0.6. We use `-O0` optimization level IRs following prior work [18]. The Linux kernels are evaluated with both `defconfig` (default configuration) and `allyesconfig` (enables almost all modules). The experiments are conducted on a machine with two Intel Xeon Gold 6248 CPUs (40C/80T) and 1TB of RAM, running Ubuntu 20.04.

As KELP and TFA are not publicly available, we compare KallGraph against another three SOTA indirect call analysis solutions, MLTA (commit `acb8f4`), SMLTA (commit `d3a707e`), and TyPM (commit `ff765f`). We also adapt Andersen's and Steensgaard's pointer analyses from SVF [24] to implement indirect call analysis. However, Andersen's analysis exceeds the 72-hour timeout for all programs except Xen-4.18, indicating its unsuitability for large-scale programs. Steensgaard's analysis finishes most of the programs in 72 hours except `allyesconfig` kernels.

7.1. Precision of KallGraph

In Table 1, the overall results show that KallGraph has consistently smaller quartiles and average numbers of indirect call targets (per icall) compared to MLTA and TyPM, meaning KallGraph achieves significant precision improvement in all target programs. In the best case, KallGraph reduces more than 90% of icall targets in `allyesconfig` Linux-5.15.

Interestingly, even though Steensgaard's analysis is known to be imprecise, it still produces fewer indirect call targets on average than MLTA for 2 of the 6 programs that it manages to analyze successfully. However, for the remaining 4, it performs significantly worse than MLTA because its unification-based equivalence classes (ECs) are easily collapsed in large-scale programs, leading to an inflation in the number of targets.

SMLTA eliminates considerable indirect call targets on average compared to MLTA (66.1% and 52.7% pruned for `allyesconfig` Linux kernels). It also outperformed TyPM in 6 out of 8 programs. However, compared to KallGraph, SMLTA eliminates more targets only for Xen. For all other programs, especially large-scale programs such as Linux, KallGraph consistently produces much smaller indirect call target sets than SMLTA, on average 63% to 73% smaller. We

Table 1. OVERALL RESULTS OF KALLGRAPH AND COMPARISON WITH SOTA METHODS

target program	method	icall w/ target[1]	Q_1^2	Q_2	Q_3	Q_4(max)	average[3]	mem	cpu· hour	time[4]
Xen-4.18 302 IRs 595 icalls	MLTA	537 (90.2%)	1	2	4	77	10.8	1GB	4s	4s
	Steensgaard	471 (79.2%)	1	2	3	70	(9.3%↓) 9.8	2GB	1h,17m	1h,17m
	SMLTA	164 (27.6%)	0	0	1	53	(85.2%↓) 1.6	1GB	4s	4s
	TyPM	525 (88.2%)	1	2	3	68	(14.8%↓) 9.2	1GB	9s	9s
	KallGraph	513 (86.2%)	1	2	2	25	(78.7%↓) 2.3	2GB	2m,00s	21s
QEMU-8.1 1,907 IRs 3,069 icalls	MLTA	2,330 (75.9%)	1	2	5	1,256	29.1	2GB	15s	15s
	Steensgaard	2,459 (80.1%)	1	2	5	960	(14.1%↓) 24.9	9GB	7h,55m	7h,55m
	SMLTA	1,413 (46.0%)	0	0	2	1,252	(47.1%↓) 15.4	2GB	14s	14s
	TyPM	2,016 (65.7%)	0	1	4	1,246	(30.6%↓) 20.2	3GB	1m,52s	1m,52s
	KallGraph	1,989 (64.8%)	0	1	3	446	(81.4%↓) 5.4	8GB	12m,25s	55s
Wine-8.16 1,574 IRs 16,666 icalls	MLTA	10,514 (63.1%)	0	1	5	2,074	12.7	3GB	52s	52s
	Steensgaard	9,883 (59.3%)	0	1	6	916	(40.2%↑) 17.8	17GB	15h,42m	15h,42m
	SMLTA	5,176 (31.0%)	0	0	2	592	(19.7%↓) 10.2	3GB	50s	50s
	TyPM	10,430 (62.6%)	0	1	4	2,074	(10.2%↓) 11.4	4GB	3m,55s	3m,55s
	KallGraph	8,490 (51.1%)	0	1	2	283	(74.8%↓) 3.2	14GB	33m,18s	1m,17s
Freebsd-14 2,381 IRs 11,163 icalls	MLTA	9,961 (89.2%)	1	2	4	1,047	15.8	5GB	37s	37s
	Steensgaard	9,039 (81.0%)	1	2	7	890	(20.9%↑) 19.1	30GB	17h,44m	17h,44m
	SMLTA	5,981 (53.6%)	0	1	4	1,041	(21.5%↓) 12.4	5GB	39s	39s
	TyPM	6,950 (62.2%)	0	2	4	1,022	(39.9%↓) 9.5	8GB	8m,45s	8m,45s
	KallGraph	7,202 (64.5%)	0	2	4	236	(79.1%↓) 3.3	22GB	2h,05m	3m,11s
Linux-5.15 `defconfig` 2,515 IRs 10,344 icalls	MLTA	8,971 (86.7%)	1	2	7	879	22.4	6GB	1m,10s	1m,10s
	Steensgaard	9,058 (87.6%)	1	3	9	1413	(86.2%↑) 41.7	34GB	48h,50m	48h,50m
	SMLTA	6,693 (64.7%)	0	2	7	874	(37.1%↓) 14.1	6GB	59s	59s
	TyPM	8,897 (86.0%)	1	2	6	879	(33.9%↓) 14.8	17GB	12m,56s	12m,56s
	KallGraph	8,911 (86.2%)	1	2	4	879	(76.8%↓) 5.2	20GB	7h,44m	8m,09s
Linux-6.5 `defconfig` 2,654 IRs 10,539 icalls	MLTA	9,003 (85.4%)	1	3	7	917	23.1	6GB	1m,18s	1m,18s
	Steensgaard	9,173 (87.0%)	1	3	10	1641	(108.7%↑) 48.2	36GB	55h,12m	55h,12m
	SMLTA	6,711 (64.5%)	0	2	7	912	(9.7%↓) 21.1	6GB	1m,10s	1m,10s
	TyPM	8,941 (84.8%)	1	3	7	917	(32.0%↓) 15.7	19GB	12m,40s	12m,40s
	KallGraph	8,959 (85.0%)	1	2	5	917	(74.9%↓) 5.8	23GB	8h,07m	8m,52s
Linux-5.15 `allyesconfig` 19,931 IRs 74,056 icalls	MLTA	71,455 (96.5%)	1	3	11	8,906	138.0	55GB	20m,06s	20m,06s
	SMLTA	44,828 (60.5%)	0	2	7	7,434	(66.1%↓) 46.8	53GB	12m,20s	12m,20s
	TyPM	71,029 (95.9%)	1	3	10	8,871	(48.5%↓) 71.1	311GB	13h,48m	13h,48m
	KallGraph	70,744 (95.5%)	1	2	6	4,819	(90.7%↓) 12.8	208GB	242h,03m	3h,37m
Linux-6.5 `allyesconfig` 22,270 IRs 80,484 icalls	MLTA	77,437 (96.2%)	1	3	12	9,846	137.5	62GB	29m,29s	29m,29s
	SMLTA	48,274 (60.0%)	0	1	8	8,321	(52.7%↓) 65.0	59GB	15m,34s	15m,34s
	TyPM	76,859 (95.5%)	1	3	11	9,813	(42.4%↓) 79.2	386GB	19h,59m	19h,59m
	KallGraph	77,046 (95.7%)	1	2	7	5,197	(87.3%↓) 17.5	246GB	270h,51m	3h,49m

[1]ratio of indirect calls that have at least one target; [2]first quartile, 25% of indirect calls have targets less than this number; [3]average target number of indirect calls, the deduction(↓) and addition(↑) rates are compared to MLTA; [4]wall-clock time, using 80 threads;

wish to point out that SMLTA results in a significantly higher fraction of icalls with empty target sets (see "icall w/ target" in Table 1). The issue is particularly prominent in various Linux kernel configurations, where SMLTA identifies targets for only 60% to 64.7% of indirect calls (iCalls), compared to over 85% achieved by other methods. In §7.2, we will evaluate the false negatives produced by SMLTA.

We now look at the distribution of the icalls by the number of targets. From the quartile results of Q1, Q2, Q3, and Q4 in Table 1, we can see that KallGraph are consistently equal or better at Q2 (median), Q3 (75th percentile), and Q4 (maximum), with fewer icall targets than prior methods. The difference is especially stark. For example, the indirect call with the largest target set (Q4) in `allyesconfig` Linux-6.5 includes 5,197 targets for KallGraph, compared to over 8,000 for prior solutions. This is likely due to the fact that existing methods would fall back to FSA for the difficult cases, and therefore resulting in imprecision for those icalls.

To understand the distribution in more detail, we visualize the results as a continuous spectrum of `allyesconfig` Linux-6.5 in Figure 8. The X-axis represents the size range of icall target sets in the log scale. The numbers are grouped into half-open intervals based on the number of icall targets. The Y-axis represents either the number of icalls (on the left associated with bars) or the cumulative number of the icall targets (on the right associated with solid lines).

Generally speaking, KallGraph finds more icalls that have targets less than 2^4=16, while TyPM and MLTA identify significantly more icalls with target sizes between $[2^6, \infty)$.

This results in more obvious differences in the solid lines, which represent the cumulative icall targets contributed by icalls with varying target set sizes. As the solid blue line shows, 91.3% MLTA targets are contributed by icalls with more than 2^8 targets; as a comparison, only 46% KallGraph targets are affected by them (solid green line). This shows that the results of MLTA are heavily dominated by those

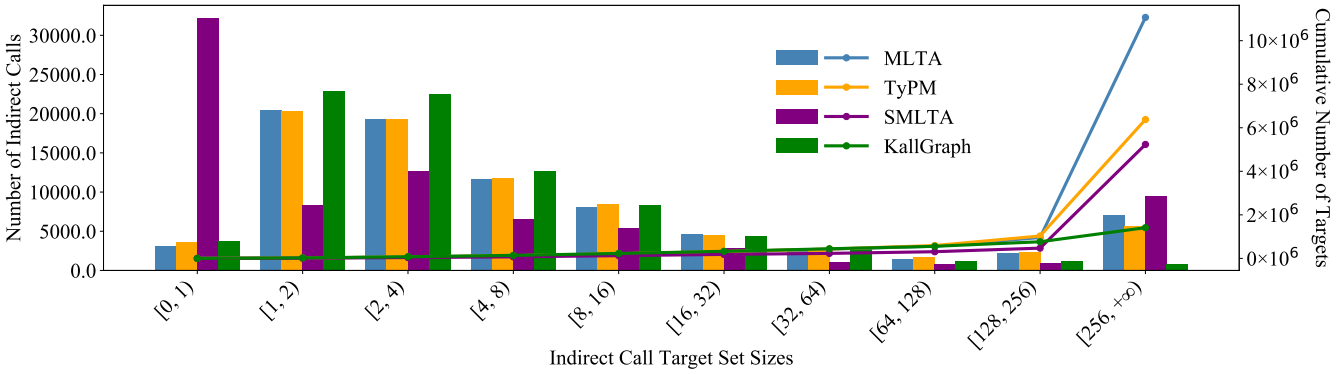

Figure 8. Distribution of KallGraph and SOTA Methods. The green bars indicate KallGraph has a greater proportion of icalls with fewer targets. In contrast, the blue solid line demonstrates that MLTA has icalls with larger target numbers, whereas icalls with fewer than 2^8 targets contribute only less than 10% to the cumulated target count.

"imprecise" icalls, likely due to its fallbacks to FSA.

We take the icall that experienced the max targets (9,846 by MLTA) as an example and illustrate its imprecision:

```
1  //drivers/misc/lis3lv02d/lis3lv02d_i2c.c:134
2    struct lis3lv02d_platform_data *pdata =
3                   client->dev.platform_data;
4    ret = pdata->setup_resources();
```

The struct layer for the icall at line 4 is {`struct lis3lv02d_platform_data, setup_resources`}, and the `struct lis3lv02d_platform_data` is cast from a `void*` field (i.e., `dev->platform_data`). This type escape forces MLTA to fall back to FSA (③ in §3), leading to nearly 10,000 potential targets. However, none of them is a true target, since this icall is defined as an interface to loadable modules whose source codes are not available in mainline Linux kernel [44]. As a comparison, KallGraph precisely handles the cast and correctly reports zero targets.

Since KELP [19] and TFA [20] are not open-sourced. Judging by the reported results in their papers, they achieve similar reductions in icall targets compared to TyPM. Specifically, KELP eliminates 54.2% targets on an unknown version of Linux kernel, and TFA eliminates 59.0% of targets on `allyesconfig` Linux-5.18. For comparison, TyPM and SMLTA eliminate 48.5% and 66.1% of targets on the `allyesconfig` configuration of Linux-5.15, respectively. Since KallGraph eliminates 90.7% of icall targets in this Linux kernel version, it roughly translates into eliminating about 70% of icall targets of KELP and TFA. Note that variations in Linux kernel versions may account for differences in results; however, we use these as approximate estimates for the purpose of comparison.

Evaluation on Optimized IRs. Some applications may require analysis on optimized IRs. To demonstrate the generality of KallGraph across different optimization levels, we evaluated both KallGraph and SOTA methods at the default optimization level (i.e., `-O2`) for Linux-5.15 and Linux-6.5, following the same setup as in Table 1. The results are summarized in Table 2.

As discussed in §3 (⑤ General Limitations and Flaws), `-O2` IRs may yield seemingly more precise results due to reduced code size and simplified control flow, but they also obscure critical program semantics (e.g., more complex instruction patterns). As a result, they are generally more difficult to handle and may produce more false negatives [45]. Despite this, the results in Table 2 show that KallGraph maintains its advantage over SOTA methods, demonstrating strong generality even on highly optimized IRs. Nevertheless, we recommend using `-O0` IRs when soundness is a priority. The evaluations in the following sections are therefore conducted on `-O0` IRs.

7.2. False Negative Analysis

We demonstrated that KallGraph effectively prunes a significant fraction of indirect call targets compared to SOTA methods. This naturally raises the critical question: does KallGraph achieve this reduction by excluding valid targets, thus introducing false negatives (FNs)?

To investigate this, we performed a comprehensive false negative analysis using the most complex and challenging Linux-6.5 as a representative target program. Specifically, we performed both dynamic tracing and manual verification covering over two thousand icalls.

Dynamic Tracing. To find FNs, one common strategy is to trace all indirect calls dynamically [11, 18–21]. To maximize the dynamic coverage, we use the state-of-the-art syscall fuzzer Syzkaller [46] to perform a 7-day fuzzing with 40 VMs on Linux-6.5, with modified QEMU-4.2.1 for tracing. Eventually, we collected 937 unique icalls with 981 targets, which we then compare against the static analysis results.

Static Analysis and Manual Verification. Even though our fuzzing campaign is extensive, the overall coverage is still limited (8.9% icalls). To find as many FNs as possible, we first isolate the differences in results between KallGraph and SOTA methods, followed by an extensive manual verification of these discrepancies. Due to the large number of icall targets, the manual verification starts from icalls instead of targets. Our manual analysis carefully tracks the struct layers and regional data flows for each icall, comparing and contrasting the execution logic of KallGraph and other methods that leads to the discrepancy, and subsequently check if possible targets exist. Eventually, our analysis encompasses

Table 2. OVERALL RESULTS ON −O2 IRS

target program	method	icall w/ target	Q_1	Q_2	Q_3	Q_4(max)	average	mem	cpu· hour	time
Linux-5.15	MLTA	7,643 (84.2%)	1	2	6	879	23.4	5GB	1m,03s	1m,03s
`defconfig`	SMLTA	4,726 (52.0%)	0	1	5	873	(37.2%↓) 14.7	5GB	57s	57s
2,515 IRs	TyPM	7,356 (81.0%)	1	2	6	879	(39.7%↓) 14.1	16GB	11m,29s	11m,29s
9,082 icalls	KallGraph	6,885 (75.8%)	1	2	4	879	(79.1%↓) 4.9	21GB	8h,11m	8m,55s
Linux-6.5	MLTA	8,037 (85.0%)	1	3	7	958	24.6	6GB	1m,10s	1m,10s
`defconfig`	SMLTA	4,913 (51.9%)	0	1	6	952	(4.9%↓) 23.4	6GB	1m,01s	1m,01s
2,654 IRs	TyPM	7,653 (80.9%)	1	3	7	958	(39.4%↓) 14.9	19GB	14m,13s	14m,13s
9,460 icalls	KallGraph	7,031 (74.3%)	0	2	4	958	(78.5%↓) 5.3	24GB	8h,53m	9m,45s
Linux-5.15	MLTA	116,541 (89.8%)	1	2	8	8,751	93.1	51GB	24m,55s	24m,55s
`allyesconfig`	SMLTA	78,721 (60.6%)	0	2	7	7,266	(42.5%↓) 53.5	52GB	11m,19s	11m,19s
19,931 IRs	TyPM	114,121 (87.9%)	1	2	7	8,693	(48.9%↓) 47.6	235GB	12h,04m	12h,04m
129,833 icalls	KallGraph	113,097 (87.1%)	1	2	5	4,190	(81.1%↓) 17.6	146GB	297h,51m	4h,05m
Linux-6.5	MLTA	125,941 (89.6%)	1	2	8	9,843	95.4	55GB	28m,31s	28m,31s
`allyesconfig`	SMLTA	87,266 (62.1%)	0	2	7	8,254	(44.9%↓) 52.6	56GB	14m,05s	14m,05s
22,270 IRs	TyPM	122,511 (87.2%)	1	2	7	9,805	(50.7%↓) 47.0	256GB	14h,21m	14h,21m
140,573 icalls	KallGraph	121,617 (86.5%)	1	2	5	4,828	(81.2%↓) 17.9	177GB	328h,40m	4h,29m

more than 2,000 icalls and 4,000 targets, requiring over 150 person-hours. To our knowledge, our manual verification is the most extensive in the space of indirect call analysis.

False Negative Result Summary. As shown in Table 3, using dynamic tracing, no FNs were identified for KallGraph, whereas MLTA, TyPM, and SMLTA all exhibited significant FNs: 18/937=1.9%, 22/937=2.3%, and 163/937=17.4% respectively. Using static analysis and manual verification, we found only 2 icalls with 9 targets being FNs of KallGraph, whereas MLTA, TyPM, and SMLTA all exhibited significantly more FNs in the hundreds of icalls and thousands of icall targets. For MLTA, we identified 100 FN icalls with 1,703 targets, of which 18 icalls with 18 targets were dynamically verified. We provide a detailed breakdown of the results in Table 6 for reproducibility (in the appendix due to space constraints). TyPM and SMLTA inherited all of MLTA's FNs and produced even more. For SMLTA, we only give a lower bound because there are too many potential FN candidates to analyze manually. In §7.2.1 and §7.2.2, we will dive into more details about how these results are produced.

Table 3. OVERALL RESULTS OF FALSE NEGATIVES

	MLTA		TyPM		SMLTA		KallGraph	
	icall	target	icall	target	icall	target	icall	target
Dynamic[1]	18	18	22	22	163	169	0	0
Static[2]	100	1,703	143	1,822	136+	1,785+	2	9

[1] FNs identified by dynamic tracing
[2] FNs identified by static analysis and manual verification

7.2.1. False Negatives of SOTA Methods. It is important to note that all successors [11, 19–21] of MLTA claim to introduce no additional FNs compared to MLTA; however, as shown in Table 3, such a claim is disproven for TyPM and SMLTA. In this section, we primarily focus on analyzing the FNs for MLTA, connecting the results to the root causes of unsoundness limitations described in §3.

Here we focus on the manual verification due to the limited coverage of dynamic tracing. To gain a more complete understanding of FNs, we calculate the discrepancies between KallGraph and the SOTA methods, as shown in Table 4, and treat these discrepancies as candidate FNs for the SOTA methods. For `defconfig` Linux-6.5 (the first row) specifically, there are 115 icalls that KallGraph finds at least one target whereas MLTA finds zero. Upon inspection, we found that 100 out of 115 (87.0%) icalls corresponding to 1,703 out of 1,778 (95.6%) targets are FNs of MLTA. These FNs are also inherited by TyPM and SMLTA.

Table 4. DISCREPANCY: CANDIDATE FNS FOR SOTA METHODS

	MLTA		TyPM		SMLTA	
	icall	target	icall	target	icall	target
Linux-6.5d	115[1]	1,778[2]	172	1,927	1,729	8,907
Linux-6.5a	1,939	13,310	2,490	22,180	21,193	181,380

[1] The first two columns: For 115 icalls, KallGraph finds at least one target for each icall (1,778 targets in total[2]), while MLTA finds zero target

After thoroughly going through all these 100 FNs, we provide a breakdown of reasons (also list in Table 6) we previously laid out in §3 and some case studies:

66 FNs due to the unsound type confinement (② in §3). For example, for the icall `link->doit()`, one of its target `neigh_get()` is address taken and stored to a field named `doit` of `struct rtnl_link` inside function `rtnl_register`. However, with an intra-procedural analysis, the struct layer cannot be found by MLTA, and the function is thus mistakenly excluded from the icall target.

```
1  //net/core/rtnetlink.c:6445
2    err = link->doit(skb, nlh, extack);
3  //net/core/neighbour.c:3876
4    rtnl_register(..., neigh_get, ...);
```

15 FNs are because of the unsound type cast handling (④ in §3). For example, global variable `icx_uncore_iio` is compiled as an anonymous struct type in LLVM IRs. Such anonymous type will introduce type casts at its use sites, which unfortunately is missed by MLTA. Differently, KallGraph's individualized type analysis in KallGraph precisely tracks the exact cast instruction.

19 FNs are due to mishandling of LLVM IRs (⑤ in §3), even if the struct layers of both icalls and address-taken functions could be ideally found intra-procedurally, but MLTA neither identifies nor falls back to FSA. We list the icall example which has the most targets here:

```
1  //arch/x86/entry/common.c:112
2    ia32_sys_call_table[unr](regs);
3  //arch/x86/entry/common.c:50
4    sys_call_table[unr](regs);
```

This icall in line 2 invokes a 32-bit system call from the `ia32_sys_call_table` function pointer array. However, MLTA fails to find any target because it does not handle arrays properly in its implementation. More importantly, since there are no struct layers under the context, in the best case, MLTA could only soundly handle it by falling back to FSA, which will introduce FPs such as 64-bit system calls (i.e., targets of `sys_call_table`) since system calls share the same signature. As a comparison, KallGraph does not have this issue since it properly handles arrays (§6) with SVF PAG, which has already parsed and represented related instructions into corresponding edges.

For the remaining cases – 15 out of 115 icalls and 78 out of 1,778 targets, we find they are imprecise cases (i.e., false positives) of KallGraph. The reason is that KallGraph follows spurious dataflow introduced by type analysis that should not have existed, leading to more targets.

In Table 4, we see that TyPM has a moderate increase in candidate FNs. In the end, we verified that 143 of 172 TyPM icalls and 1,822 of its 1,927 icall targets are verified to be FNs. In contrast, SMLTA exhibits 1,729 candidate FNs in icalls and 8,907 icall targets. Unfortunately, it is impractical to perform manual verification through all these candidate icalls. Nevertheless, we can confirm that SMLTA has at least 136 FN icalls because they appear as FNs in TyPM also. Therefore, we list 136+ as the Static results for SMLTA in Table 3. It is likely that there are many more FNs, as suggested by the heightened FNs from the dynamic tracing results. Upon reviewing the SMLTA source code, we found that SMLTA's "strong multi-layer" implementation underestimates the complexity of analyzing raw LLVM IRs. It refines icalls too aggressively in scenarios that it should fall back, resulting in a significant number of FNs.

Finally, note that the breakdown above is for the `defconfig` Linux-6.5. As shown in Table 4, if we look at the `allyesconfig` Linux-6.5 (the second row), there are 10+ times more potential FNs for SOTA methods.

7.2.2. False Negatives of KallGraph. We have shown how KallGraph effectively reveals FNs of SOTA methods. In this section, we look for FNs of KallGraph itself.

Opposite to what we analyzed in Table 4, we examine icalls where KallGraph identifies no targets, but MLTA finds at least one (we use MLTA as the baseline here because it produces the least FNs in SOTA methods). This leads to a total of 152 icalls. After going through all of them, we found only 2 icalls corresponding to 9 targets are FNs of KallGraph, as follows:

```
1  //drivers/usb/host/ohci-pci.c:251
2    int (*quirk)(struct usb_hcd *ohci);
3    quirk = (void *)quirk_id->driver_data;
4    ret = quirk(hcd); // ohci_quirk_ns()
5  //arch/x86/power/cpu.c:472
6    pm_cpu_match_t fn;
7    fn = (pm_cpu_match_t)m->driver_data;
8    ret = fn(m); // msr_save_cpuid_features()
```

Both are caused by the fact that the field `.driver_data` is a long integer that stores pointers (it gets cast back to the function pointer in this example). This means that there are `ptrtoint` and `inttoptr` instructions in LLVM IR, which need to be handled properly so that we understand such integer fields can be part of a struct layer. Unfortunately, the SVF PAG [43] maintains the relationship between *pointers* and not *integers*, and thus misses the related edges.

At a high level, this issue could be addressed by expanding the SVF PAG to incorporate nodes and edges for integer-related operations, which we leave as future work. Additionally, we discussed an alternative solution in §8.

Despite our best efforts, "unfortunately" dynamic tracing does not reveal any FNs for KallGraph.

To gain an even more complete understanding of the false negative performance of KallGraph, we conducted an additional intensive manual verification. Specifically, we randomly sampled 2,000 address-taken functions out of 159,601 across 1,471 icalls where KallGraph finds some targets, but MLTA finds more. We followed the same manual verification procedure and took more than 100 hours to finish. In the end, we did not identify any of the address-taken functions to be FNs (instead they are all FPs of MLTA).

In summary, KallGraph is substantially superior to SOTA methods regarding false negative performance.

7.3. Scalability of KallGraph

The scalability result is shown in Table 1, where we list the max memory consumption, CPU hours, and wall-clock time using 80 threads for each analysis. As mentioned in §5.4, KallGraph requires an iterative analysis approach. In our evaluations, the call graph reached a fixed point within four rounds for all target programs.

As we can see in Table 1, KallGraph does take significantly more CPU hours due to its more precise nature. In the most challenging target of `allyesconfig` Linux-6.5, it takes 270 CPU hours. Nevertheless, since KallGraph is highly parallelizable, with 80 threads, KallGraph finishes the analysis in less than 4 hours of wall-clock time. When compared to TyPM, KallGraph finishes the analysis more quickly (in wall time) and has less memory consumption.

It is worth noting that the high memory usage stems from the construction of SVF PAG, which is a prerequisite of our hybrid analysis. Once constructed, KallGraph accesses the PAG in a read-only manner, consuming less than 30MB of memory per thread. This lightweight memory consumption, combined with the highly parallelizable design of KallGraph, enables faster analysis when more CPU cores are available. In contrast, SOTA methods typically include one or more pre-processing phases that construct lookup maps for subsequent queries. These phases are difficult to parallelize due to inter-leaved read/write operations and complex data dependencies, which was also discussed as a limitation in prior work [11].

7.4. Ablation Study

In this section, we present the ablation study regarding the scalability and precision improvement by Optimized Fixed-Point Algorithm (§5.4) and Optimized CastMap (§5.5). There are three configurations, (1) Baseline (naive fixed-point algorithm + original CastMap), (2) B + F (optimized fixed-point algorithm + original CastMap), and (3) B + F + C (optimized fixed-point algorithm + optimized CastMap).

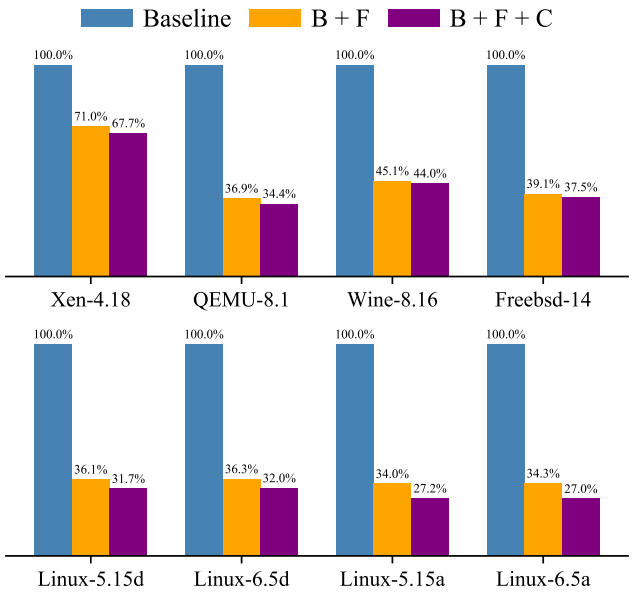

Figure 9. Scalability performance of three configurations. We compare the CPU hours and normalize the results, with the Baseline set to 100%.

Figure 9 shows the scalability results. We can see the optimized fixed-point algorithm improves scalability by 29% to 66% compared to the baseline. The optimized CastMap provides an additional 3.3% to 7.3% improvement.

Since the optimized fixed-point algorithm does not impact precision, we report the precision improvement by comparing the Baseline with B + F + C in Table 5. We can see a marginal improvement as the optimized CastMap decreases the search space of potential aliases (some of which are false positives).

Table 5. Ablation Study of Precision Improvement

	Xen-4.18	QEMU-8.1	Wine-8.16	Freebsd-14
Baseline	2.3	5.5	3.2	3.3
B + F + C	2.3	5.4	3.2	3.3
	Linux-5.15d	**Linux-6.5d**	**Linux-5.15a**	**Linux-6.5a**
Baseline	5.4	6.0	13.1	17.8
B + F + C	5.2	5.8	12.8	17.5

Average indirect call target numbers

8. Discussion

Improvement for Soundness. As discussed in §7.2.2, KallGraph does introduce FNs, which are all due to implementation-level issues, i.e., `ptrtoint` and `inttoptr` instructions. We believe they can be incorporated into SVF PAG, which we can then design rules to handle them. Alternatively, a heuristic patch could "fall back" to FSA when encountering the `ptrtoint` instruction (casting a pointer to an integer) where the pointer's control is lost. This additional check can be performed by upcasting SVF PAG nodes to variables to detect their usage in such instructions.

Improvement for Precision. We can potentially improve the precision of KallGraph by incorporating context sensitivity [47, 48]. We can also incorporate heap clone analysis [49] that can potentially prune certain infeasible aliases. Finally, we realize that sometimes KallGraph can generate false targets that are even outside of the FSA results due to on-demand points-to analysis not verifying types by its points-to nature. Given that FSA is believed to be the soundest indirect call analysis, we can easily refine the results of KallGraph by filtering any targets that are outside of the FSA results.

Improvement for Scalability. Given that our work operates on a graph, we can apply classic graph pre-processing techniques on the SVF PAG such as cycle elimination [50]. In addition, we can perform caching of the part of the graph that has already been traversed, and leverage existing graph processing frameworks that scale well to extremely large graphs on a single machine [48, 51].

Downstream Applications. While we do not directly evaluate specific downstream applications, the standard metrics directly mirror the core requirements of many downstream use cases. For instance, in control-flow integrity (CFI), improved soundness (fewer FNs) ensures legitimate control transfers are preserved, while higher precision (fewer FPs) reduces the attack surface by avoiding unnecessary target over-approximation. In bug detection, overly imprecise call graphs can lead to significant false alarms, and reducing FPs in bug detection is generally desirable. Exploring these applications is a promising direction for future work.

9. Related Work

Call Graph Analysis. In addition to MLTA [18], and its successors [11, 19–21], which we discussed extensively. We now describe other related work in the space. On-the-fly call graph construction has been studied and described in many prior works [18, 52–59]. Broadly, call graph construction techniques can be divided into points-to-based analysis and type-based analysis. For points-to-based analysis, some static program analyses [3, 7, 60] build call graphs on their own, following a variety of variants of Andersen's rules, and are not aiming for soundness. Unification-based points-to analysis [23, 55] build call graph in a fast but imprecise way. [56, 57] selectively use points-to analysis to optimize precision for a subset of icalls and targets. For type-based analysis, FSA [17, 28] was commonly known and used as primary call graph analysis for decades. [58] improves type inference methods to fix type propagation in FSA.

Applications of Call Graphs. The call graph is widely used in static program analysis. May static bug finding

techniques [5, 10, 12, 15, 59, 61–66] utilize existing call graph analysis such as MLTA. [63, 65] mentions that its results can be improved if a more precise call graph exists. [10] claims it prefers an unsound but precise points-to-based call graph analysis from [60] since MLTA introduces too many FPs that bloat strongly-connected (SCC) components in the call graph in their algorithms. Program hardening techniques such as control-flow integrity [1, 2, 14] and compartmentalization [67, 68], generally require indirect call analysis to generate sound access control rules. Otherwise, the program may fail unexpectedly due to incorrect access control policy.

10. Conclusion

This paper presents a systematic and in-depth study of state-of-the-art indirect call analyses, revealing their fundamental limitations in soundness and precision, including those not known in the original papers. Based on the insights, we formalize a framework that combines on-demand points-to analysis with type-based analysis. Finally, we developed KallGraph based on the framework and achieved results that substantially outperform state-of-the-art methods.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and valuable suggestions. This material is based upon work supported by the National Science Foundation under Grant No. #2155213 and the United States Air Force and DARPA under Agreement No. FA8750-24-2-0002.

References

[1] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *USENIX Security*, 2019, pp. 195–211.

[2] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "{In-Kernel}{Control-Flow} integrity on commodity {OSes} using {ARM} pointer authentication," in *USENIX Security*, 2022, pp. 89–106.

[3] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "{DR}.{CHECKER}: A soundy analysis for linux kernel drivers," in *USENIX Security*, 2017, pp. 1007–1024.

[4] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, "Automated partitioning of android applications for trusted execution environments," in *ICSE*, 2016, pp. 923–934.

[5] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *ACM CCS*, 2020, pp. 1165–1184.

[6] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, "Slimium: debloating the chromium browser with feature subsetting," in *ACM CCS*, 2020, pp. 461–476.

[7] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *ACM CCS*, 2021, pp. 811–824.

[8] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen, and Q. He, "Detecting missed security operations through differential checking of object-based similar paths," in *ACM CCS*, 2021, pp. 1627–1644.

[9] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *USENIX Security*, 2021, pp. 2041–2058.

[10] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger *et al.*, "Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel," in *NDSS*, 2022.

[11] K. Lu, "Practical program modularization with type-based dependence analysis," in *IEEE SP*, 2023, pp. 1256–1270.

[12] Q. Wu, Y. Xiao, X. Liao, and K. Lu, "{OS-Aware} vulnerability prioritization via differential severity analysis," in *USENIX Security*, 2022, pp. 395–412.

[13] G. Li, H. Zhang, J. Zhou, W. Shen, Y. Sui, and Z. Qian, "A hybrid alias analysis and its application to global variable protection in the linux kernel," in *USENIX Security*, 2023, pp. 4211–4228.

[14] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the effectiveness of type-based control flow integrity," in *ACSAC*, 2018, pp. 28–39.

[15] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *ICSE*, 2018, pp. 327–337.

[16] A. Milanova, A. Rountev, and B. G. Ryder, "Precise call graphs for c programs with function pointers," *ASE*, 2004.

[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, techniques, and tools," 2006.

[18] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *ACM CCS*, 2019, pp. 1867–1881.

[19] Y. Cai, Y. Jin, and C. Zhang, "Unleashing the power of type-based call graph construction by using regional pointer information," in *USENIX Security*, 2024.

[20] D. Liu, S. Ji, K. Lu, and Q. He, "Improving {Indirect-Call} analysis in {LLVM} with type and {Data-Flow}{Co-Analysis}," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5895–5912.

[21] T. Xia, H. Hu, and D. Wu, "{DEEPTYPE}: Refining indirect call targets with strong multi-layer type analysis," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5877–5894.

[22] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.

[23] B. Steensgaard, "Points-to analysis in almost linear time," in *POPL*, 1996, pp. 32–41.

[24] SVF Framework. https://github.com/SVF-tools/SVF.

[25] Y. Sui and J. Xue, "Svf: interprocedural static value-

flow analysis in llvm," in *CC*, 2016, pp. 265–266.
[26] Y. Lei and Y. Sui, "Fast and precise handling of positive weight cycles for field-sensitive pointer analysis," in *SAS*. Springer, 2019, pp. 27–47.
[27] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *FSE*, 2016, pp. 460–473.
[28] LLVM CFI. https://clang.llvm.org/docs/ControlFlowIntegrity.html.
[29] B. Niu and G. Tan, "Modular control-flow integrity," in *PLDI*, 2014, pp. 577–587.
[30] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}," in *USENIX Security*, 2014, pp. 941–955.
[31] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," *ACM SIGPLAN Notices*, pp. 242–256, 1994.
[32] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *IEEE SP*, 2023, pp. 3262–3278.
[33] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "{StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing," in *USENIX Security*, 2022, pp. 3273–3289.
[34] H. Han, A. Wesie, and B. Pak, "Precise and scalable detection of {Use-after-Compacting-Garbage-Collection} bugs," in *USENIX Security*, 2021, pp. 2059–2074.
[35] D. Zhan, Z. Yu, X. Yu, H. Zhang, L. Ye, and L. Liu, "Securing operating systems through fine-grained kernel access limitation for iot systems," *IEEE Internet of Things Journal*, no. 6, pp. 5378–5392, 2022.
[36] C. Liu, Y. Chen, and L. Lu, "Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel." in *NDSS*, 2021.
[37] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for java," *ACM SIGPLAN Notices*, pp. 59–76, 2005.
[38] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," *ACM SIGPLAN Notices*, pp. 387–400, 2006.
[39] G. Balatsouras and Y. Smaragdakis, "Structure-sensitive points-to analysis for c and c++," in *SAS*. Springer, 2016, pp. 84–104.
[40] T. Reps, "Program analysis via graph reachability," *Information and software technology*, pp. 701–726, 1998.
[41] The Often Misunderstood GEP Instruction. https://llvm.org/docs/GetElementPtr.html.
[42] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, 2015.
[43] SVFIR or Program Assignment Graph (PAG). https://github.com/svf-tools/SVF/wiki/Technical-documentation#12-svfir-or-program-assignment-graph-pag.
[44] https://elixir.bootlin.com/linux/v6.5/source/Documentation/leds/leds-lp5521.rst#L102.
[45] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. A. Sani, "Demystifying the dependency challenge in kernel fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022.
[46] Syzkaller. https://github.com/google/syzkaller.
[47] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Trans. Program. Lang. Syst.*, 2020.
[48] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *ASPLOS*, 2017.
[49] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN Notices*, pp. 278–289, 2007.
[50] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *PLDI*, 1998, pp. 85–96.
[51] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *EuroSys*, 2017.
[52] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *ICSE*, 2020, pp. 1049–1060.
[53] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *CC*. Springer, 2003, pp. 153–169.
[54] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias analysis for object-oriented programs," in *Aliasing in Object-Oriented Programming*, D. Clarke, T. Wrigstad, and J. Noble, Eds. Springer, 2013.
[55] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *PLDI*, 2007, pp. 290–299.
[56] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for c++," in *ISSTA*, 2017, pp. 329–340.
[57] Y. Cai and C. Zhang, "A cocktail approach to practical call graph construction," *OOPSLA*, pp. 1001–1033, 2023.
[58] M. Bauer, I. Grishchenko, and C. Rossow, "Typro: Forward cfi for c-style indirect function calls using type propagation," in *ACSAC*, 2022, pp. 346–360.
[59] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "{PeX}: A permission check analysis framework for linux kernel," in *USENIX Security*, 2019, pp. 1205–1220.
[60] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with {KINT}," in *OSDI*, 2012, pp. 163–177.
[61] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: scalable path-sensitive memory leak detection for millions of lines of code," in *ICSE*, 2019, pp. 72–82.
[62] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "Grebe: Unveiling exploitation potential for linux kernel bugs," in *IEEE SP*, 2022, pp. 2078–2095.

[63] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating vulnerability assessment through poc migration," in *ACM CCS*, 2021, pp. 3300–3317.
[64] Y. Zhai, Y. Hao, H. Zhang, D. Wang, C. Song, Z. Qian, M. Lesani, S. V. Krishnamurthy, and P. Yu, "Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel," in *FSE*, 2020, pp. 221–232.
[65] T. Kim, V. Kumar, J. Rhee, J. Chen, K. Kim, C. H. Kim, D. Xu, and D. J. Tian, "{PASAN}: Detecting peripheral access concurrency bugs within {Bare-Metal} embedded applications," in *USENIX Security*, 2021, pp. 249–266.
[66] H. Zhang, J. Kim, C. Yuan, Z. Qian, and T. Kim, "Statically discover cross-entry use-after-free vulnerabilities in the linux kernel," in *NDSS*, 2025.
[67] Y. Huang, V. Narayanan, D. Detweiler, K. Huang, G. Tan, T. Jaeger, and A. Burtsev, "KSplit: Automating device driver isolation," in *OSDI*, 2022, pp. 613–631.
[68] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakc," in *NDSS*, 2022.

Appendix A. Rules of Unias and KallGraph

We provide the original rules of Unias for reference:

$$\mathbf{F} \rightarrow (\ Assign \mid Store\ \textbf{I-Alias}\ Load\ )* \quad (1)$$
$$\overline{\mathbf{F}} \rightarrow (\ \overline{Assign} \mid \overline{Load}\ \textbf{I-Alias}\ \overline{Store}\ )* \quad (2)$$
$$\textbf{I-Alias} \rightarrow \overline{\mathbf{F}}\ \textbf{I-Alias}\ \mathbf{F} \quad (3)$$
$$\textbf{I-Alias} \rightarrow \overline{Load}\ \textbf{I-Alias}\ Load \quad (4)$$
$$\textbf{I-Alias} \rightarrow \varepsilon \quad (5)$$
$$\textbf{I-Alias} \rightarrow \overline{Gep_{t,i}}\ \textbf{I-Alias}\ Gep_{t,i} \quad (6)$$
$$\textbf{I-Alias} \rightarrow Gep_{t,i}\ \textbf{I-Alias}\ Gep_{t,i} \quad (7)$$
$$\textbf{I-Alias} \rightarrow \overline{Gep_{t,i}}\ \textbf{I-Alias}\ \overline{Gep_{t,i}}\ \textbf{I-Alias} \quad (8)$$
$$\textbf{I-Alias} \rightarrow\ (Gep_{t,v} \mid \overline{Gep_{t,v}})\ \textbf{I-Alias}\ Gep_{t,i} \quad (9)$$
$$\textbf{I-Alias} \rightarrow\ (Gep_{t,v} \mid \overline{Gep_{t,v}})\ \textbf{I-Alias}\ \overline{Gep_{t,i}}\ \textbf{I-Alias} \quad (10)$$
$$\textbf{I-Alias} \rightarrow\ (Gep_{t,i} \mid \overline{Gep_{t,i}})\ \textbf{I-Alias}\ Gep_{t,v} \quad (11)$$
$$\textbf{I-Alias} \rightarrow\ (Gep_{t,i} \mid \overline{Gep_{t,i}})\ \textbf{I-Alias}\ \overline{Gep_{t,v}}\ \textbf{I-Alias} \quad (12)$$
$$\textbf{I-Alias} \rightarrow Shortcut_f \quad (13)$$
$$\textbf{I-Alias} \rightarrow Shortcut_{c-t,i}\ \textbf{I-Alias}\ Gep_{t,i} \quad (14)$$

Rules (1)–(5) represent the standard on-demand Andersen's points-to analysis. Rules (6)–(12) extend the analysis to a field-sensitive variant by incorporating field accesses in the LLVM IR (i.e., `GetElementPtr` instructions). Finally, rules (13)–(14) introduce type-based shortcuts that leverage struct types and field offsets to optimize the analysis.

After applying comprehensive CFL-Rechability rules (§5.3), the complete rules for KallGraph are as follows:

$$\mathbf{F} \rightarrow (\ Assign \mid Gep_{\ t,o} \mid \overline{Gep_{\ t,-o}} \mid Store\ \textbf{I-Alias}\ Load\ )* \quad (1)$$
$$\overline{\mathbf{F}} \rightarrow (\ \overline{Assign} \mid Gep_{\ t,o} \mid \overline{Gep_{\ t,-o}} \mid \overline{Load}\ \textbf{I-Alias}\ \overline{Store}\ )* \quad (2)$$
$$\textbf{I-Alias} \rightarrow \overline{\mathbf{F}}\ \textbf{I-Alias}\ \mathbf{F} \mid \overline{Load}\ \textbf{I-Alias}\ Load \mid \varepsilon \quad (3)$$
$$\textbf{I-Alias} \rightarrow Shortcut_f \quad (4)$$
$$\textbf{I-Alias} \rightarrow Shortcut_{c-t,o}\ \textbf{I-Alias}\ Gep_{t,o} \quad (5)$$

We use o instead of i in $Gep_{t,o}$ edges to represent using the byte offsets instead of the field indices (§6).

Table 6. SAMPLE FALSE NEGATIVES OF SOTA METHODS

icall	repeat[1]	FNs	R1[2]	R2	R3	sample true targets (*italicized targets* are dynamically verified)
arch/x86/entry/common.c:112	2	419			✔	32-bit syscalls
arch/x86/entry/common.c:50	1	348			✔	64-bit syscalls
net/core/rtnetlink.c:6445	2	52	✔			*inet6_rtm_newaddr*, neigh_get, inet_rtm_newroute
io_uring/io_uring.c:1867	1	45			✔	io_accept, io_recv, io_statx
io_uring/io_uring.c:2221	1	41			✔	io_recvmsg_prep, io_send_zc_prep, io_fgetxattr_prep
net/core/rtnetlink.c:4012	1	29	✔			*inet_dump_ifaddr*, inet_dump_fib, inet_netconf_dump_devconf
drivers/gpu/drm/drm_managed.c:74	1	14	✔			drmm_encoder_alloc_release, uncore_unmap_mmio, drm_gem_init_release
drivers/acpi/acpica/evregion.c:292	1	13	✔			acpi_ec_space_handler, i801_acpi_io_handler, i2c_acpi_space_handler
kernel/async.c:127	1	13	✔			do_scan_async, async_suspend_late, async_suspend
io_uring/io_uring.c:380	1	11			✔	io_open_cleanup, io_link_cleanup, io_xattr_cleanup
net/core/skbuff.c:988	1	10	✔			*netlink_skb_destructor*, tpacket_destruct_skb, unix_destruct_scm
drivers/acpi/bus.c:1074	1	10			✔	acpi_bus_attach, check_offline, match_any
net/ipv4/icmp.c:834	1	9			✔	xfrm4_esp_err, tunnel64_err, udplite_err
kernel/trace/ring_buffer.c:1196	1	9	✔			*trace_clock_global*, trace_clock_local, trace_clock_jiffies
drivers/gpu/drm/virtio/virtgpu_vq.c:235	1	7	✔			virtio_gpu_cmd_unref_cb, virtio_gpu_cmd_capset_cb, virtio_gpu_cmd_get_edid_cb
io_uring/io_uring.c:1783	1	7			✔	io_send_prep_async, io_readv_prep_async, io_writev_prep_async
net/ipv6/icmp.c:867	1	7			✔	udpv6_err, icmpv6_err, tcp_v6_err
arch/x86/events/intel/uncore.c:1036	3	6		✔		icx_iio_set_mapping, skx_iio_set_mapping, icx_upi_set_mapping
kernel/sched/sched.h:2922	2	6	✔			sugov_update_shared, sugov_update_single_freq, dbs_update_util_handler
crypto/algapi.c:74	1	6			✔	crypto_aead_free_instance, crypto_kpp_free_instance, crypto_ahash_free_instance
block/blk-mq-tag.c:362	1	6	✔			hctx_show_busy_rq, blk_mq_has_request, complete_all_cmds_iter
net/ipv6/ip6_fib.c:2187	1	6	✔			fib6_age, fib6_ifup, fib6_ifdown
security/keys/proc.c:245	1	6			✔	user_describe, keyring_describe
include/linux/skbuff.h:3120	1	6	✔			*tcp_wfree*, sock_wfree, sock_rfree
kernel/sched/core.c:6030	1	5			✔	pick_next_task_rt, pick_next_task_stop, pick_next_task_dl
drivers/input/ff-memless.c:395	1	4	✔			hid_lgff_play, lg4ff_play, ms_play_effect
block/blk-mq-tag.c:292	1	4	✔			blk_mq_check_expired, blk_mq_handle_expired, blk_mq_rq_inflight
lib/klist.c:221	3	4	✔			*klist_devices_put*, *klist_children_put*, klist_class_dev_put
kernel/trace/trace_events.c:2391	1	4		✔		kprobe_event_define_fields, eprobe_event_define_fields, uprobe_event_define_fields
drivers/gpu/drm/i915/gt/intel_gt_pm.c:206	1	4		✔		reset_finish, execlists_reset_finish, guc_reset_nop
kernel/workqueue.c:5570	1	4		✔		acpi_processor_throttling_fn, acpi_processor_ffh_cstate_probe_cpu, local_pci_probe
security/keys/keyctl.c:810	1	4			✔	user_read, dns_resolver_read, keyring_read
drivers/gpu/drm/i915/gt/intel_gt_pm.c:191	1	3		✔		guc_engine_reset_prepare, reset_prepare, execlists_reset_prepare
mm/mempool.c:355	7	3	✔			mempool_kfree, mempool_free_slab, mempool_free_pages
drivers/char/agp/intel-gtt.c:607	1	3		✔		i810_setup, i830_setup, i9xx_setup
drivers/acpi/acpica/nsalloc.c:97	1	3	✔			acpi_scan_drop_device, acpi_bus_private_data_handler, acpi_nondev_subnode_tag
kernel/umh.c:100	1	3	✔			umh_keys_init, init_linuxrc, umh_pipe_setup
sound/core/control.c:1986	1	3	✔			snd_hwdep_control_ioctl, snd_pcm_control_ioctl, snd_rawmidi_control_ioctl
drivers/virtio/virtio_ring.c:582	2	2	✔			vp_notify, vp_notify_with_data
io_uring/io_uring.c:3487	4	2	✔			arch_get_unmapped_area_topdown, arch_get_unmapped_area
drivers/gpu/drm/i915/gt/uc/intel_guc.h:435	2	2		✔		gen11_enable_guc_interrupts, gen9_enable_guc_interrupts
sound/core/control_compat.c:474	1	2	✔			snd_pcm_control_ioctl, snd_hwdep_control_ioctl
drivers/gpu/drm/i915/display/intel_plane_initial.c:316	1	2			✔	skl_get_initial_plane_config, i9xx_get_initial_plane_config
security/keys/request_key.c:247	1	2			✔	nfs_idmap_legacy_upcall, call_sbin_request_key
mm/ptdump.c:109	12	1	✔			*effective_prot*
drivers/base/component.c:166	1	1	✔			i915_component_master_match
drivers/acpi/acpica/evgpe.c:717	1	1	✔			acpi_ec_gpe_handler
kernel/events/uprobes.c:869	1	1	✔			uprobe_perf_filter
drivers/char/agp/intel-gtt.c:921	1	1		✔		i830_check_flags
drivers/dma/acpi-dma.c:409	1	1	✔			acpi_dma_simple_xlate
fs/direct-io.c:647	1	1	✔			fat_get_block
drivers/md/dm-region-hash.c:382	1	1	✔			dispatch_bios
drivers/pnp/driver.c:180	2	1			✔	card_suspend
block/blk-rq-qos.c:217	1	1	✔			iolat_acquire_inflight
kernel/trace/trace_dynevent.c:481	1	1	✔			trace_kprobe_run_command
fs/ext4/fsmap.c:149	2	1	✔			ext4_getfsmap_format
lib/xarray.c:354	1	1	✔			*workingset_update_node*
include/linux/serio.h:125	1	1	✔			*i8042_kbd_write*
security/keys/keyctl_pkey.c:181	4	1		✔		query_asymmetric_key
net/netfilter/nf_conntrack_netlink.c:1978	2	1	✔			nf_ct_ftp_from_nlattr
drivers/acpi/tables.c:312	1	1	✔			acpi_parse_cfmws
fs/fs_context.c:290	1	1	✔			*shmem_init_fs_context*
kernel/cpu.c:195	1	1	✔			*page_writeback_cpu_online*
drivers/i2c/i2c-core-smbus.c:590	1	1			✔	i801_access
drivers/connector/connector.c:156	1	1	✔			cn_proc_mcast_ctl
drivers/base/firmware_loader/main.c:1162	1	1	✔			regdb_fw_cb
sum[3]		100	66	15	19	

[1] icalls that follow a similar pattern and share the same FN number, including 2 copies of inline icalls
[2] reasons for unsound cases. Note that one case might have multiple reasons, which we take the most obvious and possible one. R1: unsound type confinement rule (② in §3), R2: unsound type cast handling (④ in §3), R3: weak implementation caused **lack of soundness in practice** (⑤ in §3)
[3] accumulated for 'repeat' cases

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper presents a novel hybrid pointer analysis framework to improve the precision and soundness of indirect call analysis in large program. The paper identifies both precision and soundness issues in current state-of-the-art type-based methods for icall resolution and presents an approach that integrates on-demand pointer analysis with type-based reasoning to refine the callgraph construction. KallGraph, significantly improves accuracy by pruning up to 90% of false call targets and recovering missed calls. Because Kallgraph is extremely parallelizable, it processes the Linux kernel in a few hours, outperforming the state-of-the-art approaches.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field
- Addresses a Long-Known Issue

B.3. Reasons for Acceptance

1) In depth analysis of the limitations of state-of-the-art techniques
2) Significant improvement in precision
3) Scales to large applications such as the Linux kernel
4) Open source tool

B.4. Noteworthy Concerns

1) High CPU and memory overhead stemming from its reliance on the SVF framework.
2) Limited testing of downstream applications