Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information

Yuandao Cai Yibo Jin Charles Zhang
The Hong Kong University of Science and Technology
{*ycaibb, yjinbd, charlesz*}*@cse.ust.hk*

Abstract

When dealing with millions of lines of C code, we still cannot have the cake and eat it: type analysis for call graph construction is scalable yet highly imprecise. We address this precision issue through a practical observation: many function pointers are simple; they are not referenced by other pointers, nor do they derive their values by dereferencing other pointers. As a result, simple function pointers can be resolved with precise and affordable pointer aliasing information. In this work, we advocate KELP with two concerted stages. First, instead of directly using type analysis, KELP performs regional pointer analysis along def-use chains to early and precisely resolve the indirect calls through simple function pointers. Second, KELP then leverages type analysis to handle the remaining indirect calls. The first stage is efficient as KELP selectively reasons about simple function pointers, thereby avoiding prohibitive performance penalties. The second stage is precise as the candidate address-taken functions for checking type compatibility are largely reduced thanks to the first stage. Our experiments on twenty large-scale and popular software programs show that, on average, KELP can reduce spurious callees by 54.2% with only a negligible additional time cost of 8.5% (equivalent to 6.3 seconds) compared to the previous approach. More excitingly, when evaluating the call graphs through the lens of three various downstream clients (i.e., thread-sharing analysis, value-flow bug detection, and directed grey-box fuzzing), KELP can significantly enhance their effectiveness for better vulnerability understanding, hunting, and reproduction.

1 Introduction

Call graph (CG) construction is a fundamental and essential problem, underpinning a myriad of downstream security applications (e.g., heap error detection [14, 27, 51, 52, 68], taint analysis [5, 93], fuzzing testing [11, 18, 34], and control-flow integrity [40, 41, 57, 91]). In low-level C programs, function pointers are often employed as a powerful mechanism for producing compact and flexible code. However, by their dynamic nature, function pointers pose a critical challenge to statically resolve indirect calls for feasible targets. Even worse, despite the tremendous progress over decades, the previous approaches [4, 26, 29, 46, 50, 75, 77, 80] for CG construction are either imprecise or inefficient when scaling up to the ever-growing complexity of modern software systems.

Previous Approaches. In general, there are two basic ways to construct CGs: pointer analysis and type analysis. First, pointer analysis computes the possible values of each memory location, thereby using the function-pointer values to resolve each indirect call [29, 69, 77]. Unfortunately, due to the sophisticated memory operations, existing highly precise pointer analysis has faced difficulty in scaling up to million-line software [28, 40, 50]. For example, the recent study [28] shows that a field-sensitive and flow-insensitive Andersen's pointer analysis [29, 60] fails to construct a CG for Bind (680 KLoC) within eight hours. Moreover, another recent work [40] shows that SUPA [77, 78, 80], a state-of-the-art field-, flow-, and context-sensitive demand-driven Andersen's pointer analysis, cannot construct a CG for GCC (135 KLoC) with a twelve-hour time budget.

In contrast, type analysis has shown its promise to readily handle millions of lines of code in minutes [50, 94]. Specifically, function signature analysis (FSA) [6] identifies indirect-call targets by matching the types of function pointers with the ones of address-taken functions. On the downside, FSA is highly imprecise. For instance, the recent work [94] shows that FSA for the Linux kernel spent a few minutes constructing a highly imprecise CG with 81K callees per indirect call. The main reason for imprecision is that indirect calls with a few general parameter types (e.g., void* or char*) can potentially match a large number of infeasible callees [50]. To mitigate the precision issue,

most recently, multi-layer type analysis (MLTA) [50] additionally considers types of memory objects that hold the function pointers. However, much work [28, 29, 40] shows that type-based CG construction remains imprecise for adoption in industrial settings because pure type analysis is unaware of any pointer aliasing information.

Our Contributions. In this work, we advocate a staged and concerted approach, namely KELP, to alleviate the tricky precision-scalability dilemma. Our practical observation is that many function pointers are used simply without being referenced by other pointers or deriving their values by dereferencing other pointers. Consequently, these simple function pointers can be effectively resolved by using "low-hanging" regional pointer aliasing information. Interestingly, early capturing of the unique callees invoked through these simple function pointers can help reduce the number of candidate address-taken functions for checking type compatibility.

We use the buggy program shown in Figure 1 to illustrate KELP, where the simple variable `b1` defined at Line 17 contains the address of `checked_print`, while the complex variable `b2` derives its value, the address of `unchecked_print`, by dereferencing pointer `p` at Line 18. By merely checking type compatibility, MLTA [50] can imprecisely compute two callees for both the indirect calls at Lines 13 and 14. In contrast, by using regional (not whole-program) pointer analysis to compute the def-use relations of the variable `b1` at Line 13, KELP can infer that only the function `checked_print` is invoked. Furthermore, `checked_print`, whose address is assigned only to `b1`, cannot be invoked by accessing another variable `b2` at Line 14. As a result, the type analysis can only match `unchecked_print` to the indirect call at Line 14. More illustrations are presented in § 2.

To validate our observation, we empirically studied the Linux kernel (§ 3). We revealed two major findings that 34.5% of indirect calls use simple function pointers (often as stack variables or global variables), and those indirect calls uniquely invoke 23.9% of address-taken functions, which strongly imply KELP's effectiveness.

As depicted in Figure 2, KELP has two concerted stages, arming with several customized techniques (§ 4).

- First, KELP resolves the callees of those indirect calls through simple function pointers. To this end, we present a def-use analysis, computing the regional pointer aliasing information related to the simple pointers. For each indirect call of interest, our analysis tracks the transitive def-use relations across functions in a forward and backward manner, thereby reaching the sites of address-taken functions and precisely capturing the feasible callees.
- Second, KELP resolves the callees of the remaining indirect calls through complex function pointers (i.e., relative to simple ones), resorting to checking type compatibility. Specifically, our analysis recognizes and safely removes those address-taken functions that only those indirect calls computed at the first stage can invoke. As a result, the number of the candidate functions that can be matched in the type analysis is significantly reduced.

```
1  typedef void (*fptr_t)(int*);
2  struct A {fptr_t printer;};
3  struct B {struct A a;};
4  void checked_print(int* val){
5     if(val) printf("%d\n", *val);
6  }
7  void unchecked_print(int* val){
8     printf("%d\n", *val);
9  }
10 void process_input(struct B b1, struct B b2){
11    // return NULL for invalid inputs
12    int* ret = examine_input(user_input);
13    (*b1.a.printer)(ret); // safe
14    (*b2.a.printer)(ret); // NPD
15 }
16 void get_printers(struct B* p){
17    struct B b1={.a = {.printer = &checked_print}};
18    struct B b2 = *p; // dereferencing p
19    process_input(b1, b2);
20 }
21 int main(void) {
22     struct B* p = (struct B*) malloc(...);
23     struct B b ={.a={.printer= &unchecked_print}};
24     *p = b;
25     get_printers(p);
26     ...; // e.g., clearing the allocated memory
27 }
```

Figure 1: MLTA calculates two possible callees for each of the indirect calls, which is less precise than KELP.

Notably, KELP has three eminent merits, thereby establishing a new sweet spot between precision and efficiency in the realm of type-based CG construction.

1. Given the same indirect call, KELP can always precisely compute a subset of the callees concluded by the existing type analysis [6,26,46,50]. Specifically, our two core stages work in concert: the preceding def-use tracking precisely resolves the indirect calls through simple function pointers, thereby considerably "squeezing" the address-taken functions that can be matched in the incoming type analysis.
2. KELP remains lightweight without inducing many time overheads. Our regional def-use analysis automatically distinguishes the simple function pointers from the complex ones on-the-fly, computing the affordable pointer information in an elastic manner.
3. KELP does not induce *additional* false negatives to type-based call graph construction. Specifically, our analysis conservatively handles a few corner cases (e.g., the unknown def-use chains of simple function pointers) and safely uses type-analysis results as a fallback to refrain from any new false negatives.

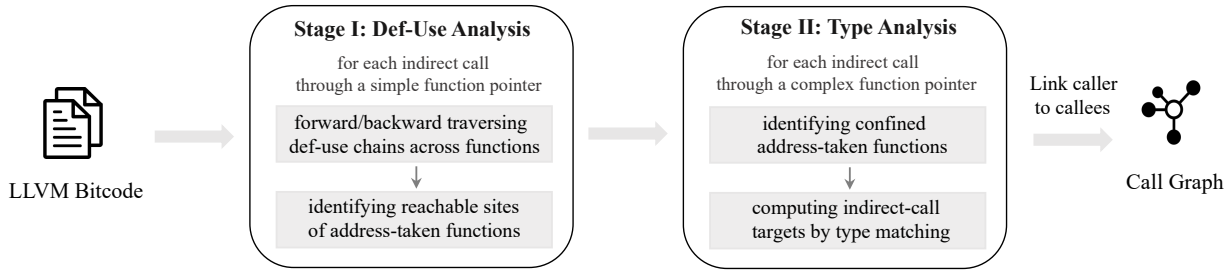


Figure 2: The workflow of KELP: a staged and concerted approach to type-based call graph construction.

To sum up, our new scheme is orthogonal to the type-analysis formulation and can serve as a pre-processing step to existing type analysis for refining indirect-call targets in a precise, efficient, and safe manner.

Experiment Highlights. We have implemented our design of KELP and performed extensive experiments on twenty software programs to demonstrate its effectiveness (§ 5 and § 6). Excitingly, even though CG construction has been studied for over forty years, KELP still can significantly advance the state-of-the-art type analysis [50] in terms of effectiveness in most programs. Furthermore, it is quite inspiring that KELP's ability to provide more precise CGs, as opposed to MLTA's, can significantly enhance three various downstream applications. Our promising results are highlighted below.

1. Compared to MLTA, on average, KELP can precisely identify 396 more uniquely-resolved indirect calls, reduce the average callee size by 54.2%, and remove 94 bogus callees from the indirect call with the largest callee size. KELP can reduce the average callee size by more than 20% for 18 programs.
2. Compared to MLTA, KELP remains lightweight and, on average, incurs only an additional 8.5% time overhead (accounting for 6.3 seconds). KELP can construct a precise CG for six million-line systems (e.g., Firefox) in around four minutes per system and for the Linux kernel in around 11 minutes.
3. KELP can help thread-sharing analysis [35, 76] to precisely reduce 25.3% more spurious thread-shared load/store memory accesses for better understanding and debugging concurrent programs.
4. KELP can help value-flow analysis, SABER [81], precisely validate three source-sink properties (i.e., memory leak, double free, and file description leak) by reducing 17.1% more false warnings on average.
5. KELP can help directed grey-box fuzzing, BEACON [34], to reproduce existing ten CVE-ID vulnerabilities by largely reducing total time costs by 51.9% and irrelevant program executions by 46.9%.

We stress that the three downstream clients broadly involve static and dynamic program analyses, as well as reasoning about sequential and concurrent programs, which can benefit various phases and tasks throughout software development. Given the importance of CGs, we believe that KELP can beef up more downstream clients. To sum up, this paper makes four main contributions:

- We advocate type analysis with regional pointer information, a big stride forward towards refining indirect-call targets in a precise and efficient way.
- We present a novel way to resolve simple and complex indirect calls in a staged and concerted manner.
- We empirically demonstrate that KELP can achieve the best precision with negligible extra time costs compared to the most recent type-based solution.
- We use the CGs produced by KELP to help improve the effectiveness of three existing clients. KELP will soon be open-sourced to foster future research.

2 KELP in a Nutshell

In this section, we first give an overview of basic pointer manipulations (§ 2.1) and then reuse the example shown in Figure 1 to show how the previous work impairs the precision of the downstream bug finding (§ 2.2). We then orchestrate the essence of KELP (§ 2.3).

2.1 Basic Concepts of Pointer Operations

We first clarify some basic terminology of pointer manipulations [30, 47, 77]. The $\&p$ denotes the address-of operation, retrieving and exposing the memory address of pointer p. In addition, we say that p is referenced by (or pointed to by) q if q contains the address of p. Particularly, when a function exposes its address (assigned to a pointer) via an address-of operation, we call the function *an address-taken function* [50]. Correspondingly, address-taken sites refer to the statements (or the initializers of global variables) where the functions expose their addresses via the address-of operations.

Besides, for the load statement $p = *q$, we say p derives the value by dereferencing q or the value of p is loaded from the memory object referenced by q. Sim-

ilarly, for the store statement $*q = p$, the value of p is stored in the memory object referenced by q.

2.2 Previous Type-Based Approaches

Much recent work [50,94] shows that the type-based call graph construction is scalable to millions of lines of code in minutes. However, existing type analysis is still highly imprecise, creating a significant obstacle to adoption in practice. We reuse the buggy code shown in Figure 1 to illustrate further that the spurious callees computed by the previous type-based approaches [6, 26, 46, 50] compromise the precision of a downstream bug checker.

Recall that the code in Figure 1 defines two structures, `A` at Line 2 and `B` at Line 3, where `A` contains a field, `printer`, with a function-pointer type, `fptr_t`, and `B` has an instance of `A`. Also, the variable `b1` is defined at Line 17, holding `checked_print`'s address. Comparatively, `b2` contains `unchecked_print`'s address, which is derived by dereferencing `p` at Line 18. Both `b1` and `b2` are passed into the function `process_input` as parameters. At Line 12, `process_input` processes the untrusted `user_input` by running the function `examine_input`. When `ret` is NULL due to the invalid `user_input`, the code can invoke `unchecked_print` at Line 14, dereferencing a NULL pointer at Line 8.

Previous Type Analysis. All the previous type analyses [6, 26, 46, 50] can imprecisely identify two callees for the two indirect calls at Lines 13 and 14. First, function signature analysis [6] (FSA) identifies indirect-call targets by matching the types of function pointers with the ones of address-taken functions. In particular, the indirect call at Line 14 uses the function pointer `b2.a.printer` with the type `fptr_t`. Since both functions, `checked_print` and `unchecked_print`, have the matched types, and their addresses are exposed (via the address-of operation) during the initialization of variables `b` and `b1`, FSA can regard both as feasible callees for Line 14. Similarly, FSA can match both functions to the indirect call at Line 13 as callees imprecisely.

To improve the precision of FSA, multi-layer type analysis [50] (MLTA) refines indirect-call targets by additionally considering the types of memory objects holding function pointers. For instance, the function pointer `b2.a.fptr_t` at Line 14 has the type `fptr_t`, and the pointer's value is loaded from the type-`A` object `a`, which is then loaded from the type-`B` object `b`. MLTA considers that the callees at Line 14 should have their addresses taken to some pointers with the three-layer type `B.A.fptr_t` rather than merely the one-layer type `fptr_t` in FSA. However, by examining Line 17, the function `checked_print` is still type-compatible to be invoked at Line 14, which is still imprecise as FSA.

Incurring False Warnings. The imprecision of CGs can compromise the effectiveness of downstream applications. Consider that a static Null pointer dereference checker uses the indirect-call results of either FSA or MLTA. The checker first finds that the `ret` at Line 12 could be NULL and proceeds to track whether the `ret` could be dereferenced somewhere without NULL checking. Along the control flows, the checker reaches the indirect call at Line 13, enters both callees, and reports an NPD warning at Line 8. However, the warning is bogus, as the indirect call at Line 13 never invokes the function `unchecked_print`. In § 2.3, we show how KELP can help the NPD checker suppress the false warning.

2.3 A Staged and Concerted Approach

We first define the basic notions, which enable KELP to improve the precision of type-based CG construction.

Definition 1. A function pointer is called *simple* when it is not referenced by other pointers and does not derive its values by dereferencing other pointers.

In essence, simple function pointers do not have the complexity introduced by referencing or dereferencing operations involving other pointers. In contrast, complex function pointers could be referenced by other pointers or obtain their values with pointer dereferencing. Correspondingly, we define an indirect call as simple (complex) when the call site uses simple (complex) function pointers to perform indirect invocation. Next, we characterize the functions that are only invoked by simple indirect calls, thus limiting the scope of address-taken functions that can be matched to complex indirect calls.

Definition 2. We refer to a function as *confined* when it can only be invoked in simple indirect calls.

That is, the address of a confined function is only propagated to simple indirect calls along def-use chains.

Back to the example in Figure 1, KELP first performs regional pointer analysis for each indirect call, tracking the def-use chains of the simple function pointer. The analysis is regional as it does not perform an exhaustive analysis of the whole program. Specifically, our analysis ensures the tracked function pointer is not referenced by other pointers and does not propagate its values with pointer dereferencing in both the forward and backward control flow paths. By tracking the def-use chains, KELP can also capture the confined values of the simple function pointer. In the example of Figure 1, since the variable `b1` is defined with the function `checked_print`'s address at Line 17 and then used at Line 13 without being referenced by other pointers or propagating its value with pointer dereferencing, KELP can precisely compute that the indirect call at Line 13 can invoke `checked_print`. In contrast, when tracing the def-use chains of the variable b2 for resolving the indirect call at Line 14, KELP

can recognize the load site at Line 18 and differentiate b2 as a complex variable that derives its value by dereferencing the pointer p. As a result, by design, KELP leaves the complex indirect call to the succeeding efficient type analysis for avoiding prohibitive performance penalties.

Next, KELP then employs the type analysis to resolve the complex indirect call at Line 14. To embrace high precision, we compute the candidate address-taken functions that can be matched due to type compatibility by removing the confined function `checked_print`. Specifically, `checked_print` is uniquely invoked by the simple indirect call at Line 13, because the function's address is taken only once and then propagated to the indirect call without pointer dereferencing. By checking the type compatibility from fewer candidate functions, as a result, KELP can precisely identify the single callee at Line 14. With the precise CG, the NPD checker can precisely identify the control flows of the bug.

Note that the previous type analyses [6, 26, 46, 50] focus on using more type information rather than reducing the candidate address-taken functions; KELP is orthogonal to their formulation and can act as their preprocessing step to further refine indirect-call targets.

Roadmap. We have introduced the essence of our staged approach to type-based call graph construction. In what follows, we orchestrate KELP in a systematic way.

1. In § 3, we perform a characteristic empirical study on the Linux kernel and reveal two major findings that significantly imply the practicality of KELP.
2. In § 4, we present a staged and concerted approach with several tailored techniques to resolve the indirect calls through simple and complex function pointers in a precise and efficient manner.

3 A Characteristic Empirical Study

To understand how prevalent simple function pointers are in practice, we conducted an empirical study on the Linux kernel (v5.15), one of the most popular open-source software. In particular, a plethora of research [22, 37, 44, 87, 88, 90] has focused on the Linux kernel owing to its vital security impacts. To reveal the effectiveness of our staged and concerted strategy, we investigate two critical research questions listed below.

- **Q1**: Is it common to use simple function pointers?
- **Q2**: Is it common for address-taken functions to be only invoked by simple indirect calls?

For Q1, if most function pointers are not simple, the regional def-use tracking at our first stage cannot effectively resolve many indirect calls. For Q2, consider that most functions are address-taken more than once and can be invoked in both simple and complex indirect calls. As

```
1  typedef void (*fptr_t)(struct unix_sock *);
2  void unix_gc(void){
3     list_for_each_entry(u, &gc_candidates, link)
4        scan_children(&u->sk, dec_inflight);
5     ...;
6     while (cursor.next != &gc_candidates){
7        scan_children(..., inc_inflight_move_tail);
8     }
9     ...;
10    list_for_each_entry(u, &gc_candidates, link)
11       scan_children(&u->sk, inc_inflight);
12 }
13 void scan_children(struct sock *x, fptr_t func){
14    if (x->sk_state != TCP_LISTEN) {
15       scan_inflight(x, func);
16    }
17 }
18 void scan_inflight(struct sock *x, fptr_t func){
19    if (test_bit(CANDIDATE, &u->gc_flags)){
20          func(u); // a simple indirect call
21    }
22 }
```

Figure 3: An example in Linux kernel (`af_unix.c`).

a result, the type analysis at our second stage cannot precisely resolve the complex indirect calls because the candidate functions for checking type compatibility cannot be largely reduced by identifying the confined functions. To answer the two questions, we statistically count each address-taken site of functions and each indirect call by analyzing the LLVM bitcode of the Linux kernel.

> **Finding I**: Considerable indirect calls (34.5%) are through simple function pointers.

More specifically, for the 87355 indirect calls being studied, many indirect calls (34.5%) are through simple function pointers. Through our observation, we have noted that simple function pointers are commonly employed as global or stack variables, where their values are directly defined during declaration. Conversely, complex function pointers tend to be implemented as heap objects. Specifically, first, the def-use chains of stack variables are very short, commonly spanning across several functions without complicated control flows. Second, the global variables are often initialized once and remain unchanged for the left execution (also called stationary [16]). To sum up, our Finding I implies that the def-use tracking at our first stage is highly cost-efficient thanks to the characteristics of simple function pointers.

Example 3.1. Figure 3 shows the usage of simple function pointers. In the function `unix_gc`, the three functions `dec_inflight`, `inc_inflight_move_tail`, and `inc_inflight`, are address-taken to the simple variable `func`, which then acts as a parameter passed from `scan_children` to `scan_inflight` and finally used in the indirect call at Line 20. Since the address of `func` is never exposed to other pointers, `func`'s value at Line

20 can be precisely determined by reasoning about direct def-use relations. If using pure type analysis for the indirect call at Line 20, a dozen false callees can be induced.

> **Finding II**: Considerable functions (23.9%) are only invoked by simple indirect calls.

In addition, for the 90127 address-taken functions being studied, most functions (76.3%) are only address-taken once, having only one address-taken site. The result implies that simple and complex indirect calls commonly invoke different groups of functions. To sum up, our Finding II implies that our type analysis at our second stage is highly effective in reducing the candidate address-taken functions for checking type compatibility.

Example 3.2. In `unix_gc` of Figure 3, the three functions, `dec_infligt`, `inc_infligt_move_tail`, and `inc_infligt`, are address-taken to and only once to the simple function pointer `func`. In other words, only the indirect call through accessing `func` at Line 20 can invoke the three functions and any other indirect calls cannot. If directly using pure type analysis [6, 50], the three functions can be matched to other type-compatible indirect calls, which can induce tremendous bogus callees.

Summary. The above two findings on the Linux kernel strongly imply the effectiveness of our two concerted stages to resolve indirect-call targets. To the best of our knowledge, the previous type-based approaches are all oblivious to simple function pointers. On the other hand, precise pointer analysis that computes the possible values of all function pointers is prohibitively expensive for large codebases [28, 41]. In § 4, we detail the key techniques behind KELP for refining indirect-call targets.

4 Algorithm Design

This section first presents the basic notions for program abstraction and states two key challenges (§ 4.1). Then, we present the two core stages in KELP (§ 4.2 and § 4.3).

4.1 Preliminaries

In this part, we describe the program abstraction for formulating our approach and state two key challenges.

Assumption. As with most previous pointer analysis and type analysis [6, 26, 46, 50, 77], we assume that the code being analyzed is closed, meaning that all of the source code for the target program is available and within the scope of our analysis. If the code is open and some functions are passed from outside the program for inner indirect calls to invoke, both the conventional pointer analysis and type analysis may not be able to resolve the corresponding indirect calls.

$$
\begin{aligned}
&\text{Program}\quad \mathbb{P} := \mathbb{F}^{+}\\
&\text{Function}\quad \mathbb{F} := f(v_1,\ldots,v_n)\{\mathbb{S}^{*};\}\\
&\text{Statement}\quad \mathbb{S} :=\\
&\qquad\qquad | \; v_1 = v_2 \mid v = \&o \mid v_1 = *v_2 \mid *v_1 = v_2\\
&\qquad\qquad | \; v_1 = \phi(v_2, v_3) \mid \mathit{return}\ v \mid v = \ v_0(v_1, v_2, \ldots)\\
&\qquad\qquad | \; v_1 = \&v_2 \rightarrow \mathit{fld} \mid s_1; s_2
\end{aligned}
$$

Figure 4: The syntax of the language.

Program Abstraction. Like much work [10, 14, 16, 30, 31, 68, 77], we formalize KELP with a standard LLVM-like language shown in Figure 4. Note that we also use the *mem2reg* [83] optimization to produce and simplify the LLVM IR of an analyzed program, which is further discussed in § 5. Specifically, we have call sites denoted as cs, which may accept a variable. We use $\mathbb{IC}$ to denote all indirect calls. The `Phi` instruction, $v_1 = \phi(v_2, v_3)$, merges the values of v_2 and v_3 from different basic blocks into a single value v_1. Besides, to reach field-sensitivity, we have the instruction that, given a memory address operand, returns a new address by adding a relative offset, corresponding to a field element.

We follow the LLVM convention of separating program variables into two disjoint sets of top-level variables ($v \in \mathbb{V}$) and address-taken objects ($o \in \mathbb{O}$). First, top-level variables may hold some memory addresses, comprising stack virtual registers and global variables. The variables $\mathbb{V}$ are in SSA form, where every variable has only one definition. Second, the objects $\mathbb{O}$ represent abstract memory locations, accessed indirectly via loads and stores by taking the variables $\mathbb{V}$ as arguments.

Points-to Relations. The points-to set of a pointer v at a statement s is denoted as $pt(s,v) = \{o | o \in \mathbb{O}\}$. In Figure 5(a), for example, we can infer $pt(s_2, p) = \{o_1\}$ at s_2. To resolve the indirect calls at s_{20} and s_{21}, KELP needs to compute $pt(s_{20}, b1)$ and $pt(s_{21}, b2)$ by identifying and tracing their def-use chains, as shown in Figure 5(b).

Def-Use Chains. We define the inter-procedural def-use graph that captures the data dependencies of variables and objects, upon which KELP resolves the simple indirect calls by the regional traversal of direct value flows.

Definition 3. A DUG is denoted as $G = (N, E)$, where

- N is a set of nodes, each of which, n, denotes a statement s. Thus, we also use s to denote a node.
- $E \subseteq N \times N$ is a set of directed edges, each of which represents a def-use relation. First, a direct value-flow edge $s_1 \xrightarrow{v} s_2$ ($v \in \mathbb{V}$) from a statement s_1 to a statement s_2 denotes a def-use relation for v with its def at s_1 and use at s_2. Second, an indirect value-flow edge $s_1 \xrightarrow{o} s_2$ ($o \in \mathbb{O}$) from s_1 (a store) to s_2 (a load or a store) denotes an approximate def-use

chain for o with its def at s_1 and use (or def) at s_2.

Remark. The direct value flows in DUG (e.g., $s_1 \xrightarrow{v} s_2$) can be readily constructed thanks to its SSA form. In contrast, the indirect value flows $s_1 \xrightarrow{o} s_2$ require additional pointer aliasing information to determine the objects of pointer dereferencing and match the loads and stores precisely. The previous pointer analyses [31, 77, 81, 89] construct the DUG by using an expensive whole-program Andersen's pointer analysis [29], which is difficult to scale up to large-scale million-line software programs (e.g., the Linux kernel) [40, 50]. Instead of tracking all value flows, KELP only precisely monitors the direct value flows to resolve simple indirect calls, which greatly reduces the performance overhead. When realizing that the tracking passes through indirect value flows with pointer dereferencing, KELP categorizes the corresponding indirect calls as complex based on Definition 1.

Example 4.1. Figure 5(b) shows the def-use chains ① from variable $b1$ defined at s_8 to $b1$ used at s_{20} and ② from variable b defined at s_3 to $b2$ used at s_{21}, which would be tracked by KELP in a backward way. For example, the $s_{18} \xrightarrow{b2} s_{21}$ is a direct edge through variable $b2$ while $s_4 \xrightarrow{o_1} s_9$ is an indirect edge through object o_1. The grey part means the corresponding chains are not backward tracked by KELP, as the value of the complex pointer $b2$ is propagated along the indirect value flow.

Problem Statement. We deliberate on two technical challenges for KELP to reach precise and efficient:

1. How to devise the def-use analysis to selectively resolve simple indirect calls and automatically avoid reasoning about complex indirect calls (§ 4.2).
2. How to leverage the regional pointer information of direct value flows to boost the precision of resolving complex indirect calls in type analysis (§ 4.3).

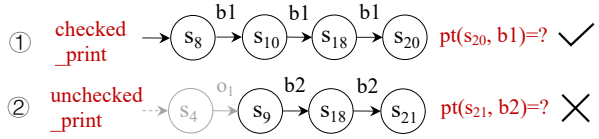
4.2 Resolving Simple Function Pointers through Efficient Def-Use Tracking

At the first stage, we resolve the indirect calls through simple function pointers (denoted as $\mathbb{IC}_1$). Specifically, we devise a regional def-use analysis to capture the values of the simple function pointers and bypass the prohibitive resolution of the complex ones. In what follows, we first investigate the def-use analysis. We then unravel the approach of resolving the indirect-call targets.

Regional Def-Use Tracking. At a high level, KELP carries out regional field-, flow-, and context-sensitive def-use analysis to resolve each simple indirect call. Specifically, the analysis ensures the function pointers being tracked are simple, thereby efficiently computing the confined values of the simple function pointers. To

```
1  int main(void){
2      p=&o1;
3      b=&unchecked_print;
4      *p=b;
5      get_printers(p);
6  }
7  void get_printers(p){
8      b1 =&checked_print;
9      b2= *p;
10     process_input(b1, b2);
11 }
12 void checked_print(int* val){
13     if (val) printf("…", *val);
14 }
15 void unchecked_print(int* val){
16     printf("…", *val);
17 }
18 void process_input(b1, b2){
19     ret = examine_input(…);
20     (*b1)(ret);
21     (*b2)(ret);
22 }
```

(a)



(b)

Figure 5: (a) shows the simplified code of Figure 1 by omitting the type information, e.g., two struct types `A` and `B`. (b) shows the tracked def-use chain (the node s_3 in the chains ② defining b is omitted) to compute the points-to set of simple $b1$ and complex $b2$, respectively.

this end, we combine both forward and backward def-use tracing for direct value flows. When perceiving that the tracked function pointers can propagate their values along the indirect value flows in either the forward or backward control-flow paths, KELP ceases further prohibitive tracking and classifies these pointers as complex.

Backward Phase. At the core, the backward analysis examines whether the function pointer of an indirect call is simple without propagating its value by dereferencing other pointers. By backward tracking the direct def-use chains, the analysis can compute the values of the interesting pointer. In detail, we conduct a backward def-use reachability analysis by computing a reachability relation denoted as $\leftsquigarrow$. Formally, $(s,p) \leftsquigarrow (s_1,q)$ signifies a computed def-use from a definition of q at s_1 to a use of p at s in DUG. Unlike the previous work [31, 77, 80], the top-level variables q and p could not be memory objects o according to the definition of direct value flows.

First, we introduce three basic rules of [FUNC-SITE], [COPY], and [PHI], as shown in Figure 6.

• In rule [FUNC-SITE] to handle $p = \&func$, a function $func$ is address-taken being referenced by pointer p, producing $pt(s,p) = \&func$. In particular, reaching the address-taken of a function means that the value of the function pointer is successfully checked.

• In rule [COPY] to handle $p = q$, a variable q is backward def-use reachable from the variable p.

• In rule [PHI] to handle $p = \phi(q,r)$, both variables q and r are backward def-use reachable from p.

Next, we describe [FIELD] to handle $p = \&q \rightarrow fld$, where a variable q is backward def-use reachable from the variable p, which is similar to [COPY]. Differently,

$$[\text{FUNC-SITE}]\ \frac{s : p = \&func}{pt(s,p) = pt(s,p) \cup \{func\}} \qquad [\text{COPY}]\ \frac{s : p = q \quad s_1 \xrightarrow{q} s}{(s,p) \leftsquigarrow (s_1,q)}$$

$$[\text{PHI}]\ \frac{s : p = \phi(q,r) \quad s_1 \xrightarrow{q} s \quad s_2 \xrightarrow{r} s}{(s,p) \leftsquigarrow (s_1,q), (s,p) \leftsquigarrow (s_2,r)} \qquad [\text{FIELD}]\ \frac{s : p = \&q \rightarrow fld \quad s_1 \xrightarrow{q} s}{(s,p) \leftsquigarrow (s_1,q) \quad p = FLD(q,fld)} \qquad [\text{LOAD}]\ \frac{s : p = *q}{(s,p) \leftsquigarrow \times}$$

$$[\text{CALL}]\ \frac{s_{call} : p = q \quad s_1 \xrightarrow{q} s_{call}}{(s_{call},p) \leftsquigarrow (s_1,q)} \qquad [\text{RET}]\ \frac{s_{ret} : p = q \quad s_1 \xrightarrow{q} s_{ret}}{(s_{ret},p) \leftsquigarrow (s_1,q)}$$

Figure 6: Basic rules for backward def-use tracking for simple indirect calls.

we introduce a new function notation $p = FLD(q, fld)$, which signifies that p represents the field fld of q. This rule is important for achieving field-sensitivity, as it enables the differentiation of various function-pointer fields and the fields' values within a struct variable.

Example 4.2. For the def-use chains ① in Figure 5(b), we can infer $(s_{20}, b1) \leftsquigarrow (s_{18}, b1) \leftsquigarrow (s_{10}, b1) \leftsquigarrow (s_8, b1)$ through the parameter passing by using the rule [COPY].

Importantly, indirect value flows refer to the propagation of values through pointer dereferencing, specifically from stores to loads. Once the tracking of def-use chains encounters these indirect value flows, the backward analysis ceases any further expensive tracking. Specifically, when the analysis reaches a load site where the value of the tracked function pointer is obtained by dereferencing other pointers, KELP employs the below [LOAD] rule.

• In rule [LOAD] to handle $p = *q$, the analysis considers that the value of p is derived by dereferencing pointer q and, thus, recognizes p as complex. Reaching a load site leads to the termination of our backward def-use analysis, indicated by the notation $(s,p) \leftsquigarrow \times$.

Example 4.3. In Figure 5(b), using [LOAD], the value of variable $b2$ at s_9 is loaded from the memory object o_1 referenced by pointer p, denoting $b2$ as a complex function pointer and s_{21} as a complex indirect call.

Moreover, we introduce the inter-procedural analysis rules (i.e., [CALL] and [RET]) for the def-use traversal. To be precise, context-sensitivity is reached by CFL-reachability [63]. Since CFL-reachability is orthogonal to our contributions, we briefly describe the process. Specifically, we maintain a string during the def-use traversal to validate the feasibility of the calling contexts. The traversed def-use path is realizable if strings between calls and returns have matched parentheses.

• In [RET] to handle a $p = q$ at s_{ret}, q is backward reachable from p. When traversing along the return edge, we append a left parenthesis $(_{cs}$ to the string.

• In [CALL] to handle $p = q$ at s_{call}, q is backward reachable from p. When traversing back to the call site, we append a right parenthesis $)_{cs}$ to the string.

Forward Phase. The goal of the forward def-use analysis is to ensure that the values of the function pointer being tracked are not propagated to memory objects referenced by other pointers in the forward control-flow path. Otherwise, the values of the function pointers may propagate to complex indirect calls, implying that the referenced values (i.e., functions' addresses) are not confined. In our second stage (§ 4.3), we would identify the confined functions that are only invoked by simple indirect calls. For the forward def-use tracking, KELP employs rules for handling statements that are similar to those used in the backward tracking. However, differently, the def-use reachability relation is computed in a forward manner.

Handling Global Variables. The tracking of global variables is different from the non-global ones. Specifically, flow-sensitively tracking def-use chains of global variables that can be assigned multiple times (without passing through explicitly as parameters and returns) is challenging, as the control flows are unknown. Thus, simple global variables are handled flow-insensitively. Specifically, when backward analyzing an indirect call and identifying the function pointer as a simple global variable, the def-use analysis also globally collects all possible ever-written values to the global variable in different regions as the potential callees. In our study of § 3, we found that most global variables of function pointers are initialized once, perhaps because multiple writes to global variables in different modules can be error-prone in large systems, thus being commonly avoided.

Resolving Simple Indirect Calls. After identifying the regional def-use relations of simple function pointers, we resolve the indirect calls. Specifically, for each indirect call $s : v = v_0(v_1, \dots)$ that is successfully tracked without termination, we identify the corresponding callees based on the reachable address-taken sites of functions and collect the indirect call in the set $\mathbb{IC}_1$.

We also record each def-use reachable address-taken site of functions after tracking (denoted as a set $DefUseReachingSites$). The address-taken sites of functions can be either the stores or the initializers of global variables. The set $DefUseReachingSites$, as shown in § 4.3, can help identify and reduce the candidate

address-taken functions for checking type compatibility, largely improving the precision of KELP.

Example 4.4. For the def-use chains ① in Figure 5(b), we can compute $pt(s_{20}, b1) = \{\&checked\_print\}$. As a result, the indirect call at s_{20} is resolved. We also collect s_8, the address-taken site of *checked_print*, for the incoming type analysis to identify the candidate functions.

Summary. By courtesy of the first stage, KELP has two salient advantages. First, our def-use tracking enables the early and principled discovery of indirect calls through simple function pointers. As a result, many indirect calls can be resolved precisely without more conservative type matching. Second, our regional def-use tracking actively avoids reasoning about the complex function pointers that could otherwise trigger performance penalties, judiciously leaving the corresponding indirect calls to the succeeding stage to embrace high efficiency.

4.3 Resolving Complex Function Pointers through Precise Type Analysis

Our second stage handles the remaining complex indirect calls (denoted as $\mathbb{IC}_2$) via type analysis, which, notably, cannot be resolved efficiently by the def-use tracking at the first stage. Specifically, instead of using the whole address-taken functions as the candidate functions for checking type compatibility, KELP removes those confined address-taken functions that only those simple indirect calls can invoke. Intuitively, confined functions are characterized by having all their address-taken sites included in the def-use chains of function pointers used in simple indirect calls. Thus, these confined functions cannot be invoked in complex indirect calls.

In what follows, we first investigate how to efficiently identify the confined address-taken functions by leveraging the reachable address-taken sites computed by the preceding def-use tracking. We then orchestrate how to remove the confined address-taken functions from the candidate functions for type checking, thereby precisely resolving the complex indirect calls.

Identifying Confined Function Set. We use *ConFunc* to denote the Confined address-taken Functions and *CandiFunc* to represent Candidate Functions for type checking. At a high level, KELP captures the functions *ConFunc*, which have all address-taken sites reachable by our forward and backward def-use tracking. Intuitively, this case guarantees that the functions' addresses are only propagated to simple indirect calls, thus only invoked by simple indirect calls. Specifically, the method `IdentifyConfinedATFunc()` in Algorithm 1 describes two critical steps to capture the functions *ConFunc*.

- Lines 2 and 7: We first check whether an address-taken function denoted by *func* can be invoked by simple indirect calls. That is, *func* is one of the callees of an indirect call *cs* in $\mathbb{IC}_1$.
- Lines 8 and 10: We next enumerate each address-taken site (denoted as *s*) of the function *func*, checking whether all the sites are def-use reachable ($s \in DefUseReachingSites$). If so, *func* is confined in simple indirect calls and put to *ConFunc*.

Algorithm 1: Precise Type Analysis.

```
1  Function IdentifyConfinedATFunc():
2      𝔽 ↩ retrieveAllFunc();
3      ConFunc ↩ ∅;
4      foreach func ∈ 𝔽 do
5          if ¬isAddressTaken(func) then
6              continue;
7          if ∃cs ∈ 𝕀ℂ1 : func ∈ ICToCalleeMap[cs] then
8              𝕊 ↩ getAllAddressTakenSites(func);
9              if ∀s ∈ 𝕊 : s ∈ DefUseReachingSites then
10                 ConFunc ↩ ConFunc ∪ {func}
11     return ConFunc
12 Function ResolveComplexICall():
13     𝕀ℂ2 ↩ 𝕀ℂ − 𝕀ℂ1;
14     ConFunc ↩ IdentifyConfinedATFunc ();
15     ATFunc ↩ getAllAddressTakenFunc();
16     CandiFunc ↩ ATFunc − ConFunc;
17     foreach cs ∈ 𝕀ℂ2 do
18         Callees ↩ retrieveTypeResult(cs) ;
19         NewCallees ↩ ∅ ;
20         foreach func ∈ Callees ∩ CandiFunc do
21             NewCallees ↩ NewCallees ∪ {func} ;
22         ICToCalleeMap[cs] = NewCallees;
23     return ICToCalleeMap;
```

Thanks to the first stage that captures all reachable address-taken sites, our second stage can efficiently recognize the confined address-taken functions *ConFunc*.

Example 4.5. Back to Figure 5(b), *checked_print* is only address-taken once, and its address-taken site s_8 is def-use reachable from $b1$ used in the indirect call at s_{20}. Thus, *checked_print* can only be invoked at s_{20}. As a result, the complex indirect call at s_{21} cannot invoke *checked_print* through accessing another pointer $b2$.

Resolving Complex Indirect Calls. After capturing the confined functions *ConFunc*, KELP identifies the candidate functions *CandiFunc*, for effectively checking type compatibility. The type analysis here can be either the traditional function signature (one-layer) analysis or the multi-layer type analysis, as identifying fewer candidate address-taken functions *CandiFunc* is orthogonal to their approach formulation. The algorithm to resolve complex indirect calls is shown by

the method `ResolveComplexICall()` in Algorithm 1. More specifically, KELP has two critical steps.

- Lines 13 and 16: We identify *CandiFunc* by finding an address-taken function absent from *ConFunc*.
- Lines 17 and 22: Finally, we retrieve the type-compatible targets for each indirect call in $\mathbb{IC}_2$ and precisely remove the callees not in the *CandiFunc*.

Example 4.6. Back to Figure 5(b), we resolve the remaining complex indirect call at s_{21}. First, we identify the candidate functions $\{unchecked\_print\}$ by removing the confined function *checked_print* from all address-taken functions $\{unchecked\_print, checked\_print\}$. Second, by checking the type compatibility, we can precisely match *unchecked_print* to the indirect call at s_{21}.

Summary. KELP confers two salient privileges over the pure type analysis [6, 50]. First, our type analysis is precise, removing those address-taken functions uniquely invoked by the simple indirect calls. As a result, the type-compatible targets of complex indirect calls are significantly refined. Second, our type analysis remains lightweight, as identifying the confined address-taken functions is efficient thanks to the def-use reachable address-taken sites provided by the preceding stage.

5 Implementation

KELP is built on the LLVM infrastructure, where it serves as the pre-processing step for the state-of-the-art multi-layer type analysis (MLTA) [50]. Figure 2 illustrates the workflow of KELP, where it takes the LLVM bitcode file of the target program as input, resolves indirect calls, and generates the call graph as output. Next, we will delve into two implementation details.

Promoting Memory to Register. LLVM IR can include excessive stack traffic for very simple and common operations, potentially degrading the performance of the def-use analysis. To address this issue, we use the mem2reg optimization pass [83], which transforms memory references into register references and introduces `Phi` nodes when necessary. Specifically, the pass promotes `alloca` instructions that are solely used for loads and stores.

Safe Fallback Strategy. We assume that the analyzed code is self-contained as a standard practice; otherwise, most previous type and pointer analysis techniques could overlook indirect-call targets. In addition, there are two possible causes of producing new false negatives by employing KELP when the def-use chains of simple function pointers cannot be fully collected. First, the def-use traversal may encounter unknown control flows or code, such as dynamically-linked libraries, indirect calls, and assembly code. Note that the case is uncommon in practice since, intuitively, the uses of simple stack variables do not span across complicated control flows. The second source is handling global variables, whose def-use chains are collected in a flow-insensitive and conservative manner. When an initialized global variable is reassigned with the value gained by dereferencing other pointers somewhere, the indirect def-use chains of the global variable are hard to be fully collected by KELP. Note that the case is uncommon because we observed that most global variables are simple and initialized once without modification anymore [16]. To mitigate both the problems, we design an automatic fallback strategy during def-use traversal in KELP. Specifically, for both the corner cases, KELP leaves the indirect calls being computed to the incoming type analysis as a backup. As a result, our approach does not introduce additional false negatives to type analysis.

6 Evaluation

This section investigates the following questions:

- § 6.1: Compared to the pure type analysis, how precise and efficient is KELP?
- § 6.2: How much can KELP improve the effectiveness of the downstream security applications?

Furthermore, we discuss other questions in § 6.3, such as the effectiveness of each stage and potential false negatives. All the experiments were finished on a computer with two 20-core Intel(R) Xeon CPU@2.20GHz and 128GB physical memory running Ubuntu-16.04.

6.1 Advancing Pure Type Analysis

First, we compare KELP to the most recent multi-layer type analysis, MLTA [50]. This experiment aims to show that our two concerted stages can effectively act as a pre-processing step to improve the pure type analysis in terms of precision with negligible extra time overheads.

Experiment Setup. Based on the four principles below, we chose twenty representative real-world C-based software suites as the benchmarks shown in Table 1. First, the software is mature, with a long code history. Second, the award-winning software has significant impacts on both academia and industry. Specifically, much work [14, 24, 50, 68, 89] has evaluated these systems to ensure their security owing to their impacts. Third, these systems represent a broad spectrum of popular applications, such as databases, network protocols, and operating systems. Fourth, the software is large-scale and extensively exploits indirect calls. Importantly, each software program has more than 100 KLoC, and seven million-line software programs are involved in showing KELP's effectiveness in industrial settings.

Table 1: Comparison on time costs (second) and precision in refining indirect-call targets.

ID	Project	Size (KLoC)	Indirect Calls	MLTA				KELP			
				Sing.	Avg.	Max.	Time	Sing.	Avg.	Max.	Time
1	OpenSSH	118	136	6	13.4	58	6.4	7 (1 ↑)	6.2 (53.7% ↓)	18 (40 ↓)	6.7 (+ 0.3)
2	GCC	135	133	10	18.7	48	8.3	11 (1 ↑)	16.1 (13.9% ↓)	48 (0 ↓)	8.7 (+ 0.4)
3	Curl	168	989	276	12.9	6	4.3	855 (579 ↑)	4.3 (66.7% ↓)	6 (0 ↓)	4.6 (+ 0.3)
4	Redis	179	383	54	23.1	105	6.8	85 (31 ↑)	15.7 (32.0% ↓)	16 (89 ↓)	7.3 (+ 0.5)
5	Git	278	507	63	22.3	153	16.2	77 (14 ↑)	13.9 (37.7% ↓)	126 (27 ↓)	16.8 (+ 0.6)
6	zfs	377	1715	58	107.3	1539	40.8	117 (59 ↑)	29.5 (72.5% ↓)	256 (1283 ↓)	42.6 (+ 1.8)
7	Vim	416	1023	20	39.2	517	14.4	899 (879 ↑)	13.2 (66.3% ↓)	517 (0 ↓)	15.0 (+ 0.6)
8	PJSIP	505	1099	207	18.0	117	19.3	294 (87 ↑)	3.1 (82.8% ↓)	117 (0 ↓)	20.1 (+ 0.8)
9	OpenSSL	513	1827	203	79.6	337	23.3	322 (119 ↑)	25.4 (68.1% ↓)	238 (99 ↓)	24.6 (+ 1.3)
10	Libicu	537	1830	67	9.8	42	31.2	142 (75 ↑)	5.1 (48.0% ↓)	42 (0 ↓)	32.4 (+ 1.2)
11	Python	560	1402	67	56.3	221	40.6	173 (106 ↑)	27.6 (51.0% ↓)	217 (4 ↓)	72.1 (+ 31.5)
12	Wrk	594	1159	217	27.8	155	11.6	326 (109 ↑)	19.7 (29.1% ↓)	155 (0 ↓)	12.7 (+ 1.1)
13	Postgres	955	1724	128	28.2	55	49.1	215 (87 ↑)	22.2 (21.2% ↓)	54 (1 ↓)	50.9 (+ 1.8)
14	FFmpeg	1213	3390	718	39.3	1443	87.5	729 (11 ↑)	35.7 (9.2% ↓)	1443 (0 ↓)	93.2 (+ 5.7)
15	PHP	1314	1737	193	36.3	1057	27.1	382 (189 ↑)	9.1 (65.4% ↓)	1029 (28 ↓)	28.7 (+ 1.6)
16	MariaDB	1769	4224	493	27.0	94	101.9	1678 (1185 ↑)	13.6 (49.6% ↓)	65 (29 ↓)	118.1 (+ 16.2)
17	MySQL	2030	3568	779	20.4	92	77.8	1398 (619 ↑)	7.4 (63.7% ↓)	87 (5 ↓)	86.6 (+ 8.8)
18	Wine	4092	12596	1205	56.0	473	76.5	3645 (2440 ↑)	16.4 (70.7% ↓)	456 (17 ↓)	79.8 (+ 3.3)
19	Firefox	7998	2738	534	12.3	233	181.1	793 (259 ↑)	7.8 (36.6% ↓)	109 (124 ↓)	201.6 (+ 20.5)
20	Linux	26181	87355	13730	12.9	443	666.3	14799 (1049 ↑)	6.8 (47.3% ↓)	296 (147 ↓)	694.1 (+ 27.8)
-	Avg.	2497	6477	951	32.5	359	74.5	1347 (**396** ↑)	14.9 (**54.2**% ↓)	265 (**94** ↓)	80.8 (+ **6.3**)

Sing., Avg., and Max. denote the number of single-callee indirect calls, the average callee size of all indirect calls, and the maximum callee size, resp.

We assess the precision of refining indirect-call targets by using three metrics, i.e., the number of single-callee indirect calls, the average callee size of all indirect calls, and the largest callee size, which are critical for various applications. In particular, uniquely-resolved indirect calls can be transformed into direct calls, subsequently inlined, and optimized further. Intuitively, more precise call graphs lead to more single-callee indirect calls and optimization. In addition, reducing the average and largest callee sizes is critical to many applications, such as CFI [40, 55, 57] and bug finding [24, 27, 52, 68].

Precision. The results are shown in Table 1, where KELP is more precise than MLTA at any metric. Compared to MLTA, on average, KELP can additionally increase 396 uniquely-resolved indirect calls, reduce the average callee size by 54.2%, and remove 94 false callees from the indirect call with the largest callee size. In addition, for 18 projects, KELP can reduce the average callee size largely by more than 20%. At most, KELP can prune away 82.8% of bogus indirect-call targets in PJSIP.

To investigate how our two concerted stages contribute to the precision of KELP, we examine the simple and complex indirect calls (i.e., $\mathbb{IC}_1$ and $\mathbb{IC}_2$), and the confined and candidate address-taken functions (i.e., *ConFunc* and *CandiFunc*). The results are shown in Figure 7. We observed that all programs use simple indirect calls, accounting for 33.0% on average, which suggests the effectiveness of computing the regional pointer information at the first stage. Also, those simple indirect calls uniquely refer to many confined functions (32.0% on average), which suggests that the candidate functions for checking type compatibility can be significantly reduced, thereby improving the precision of pure type analysis.

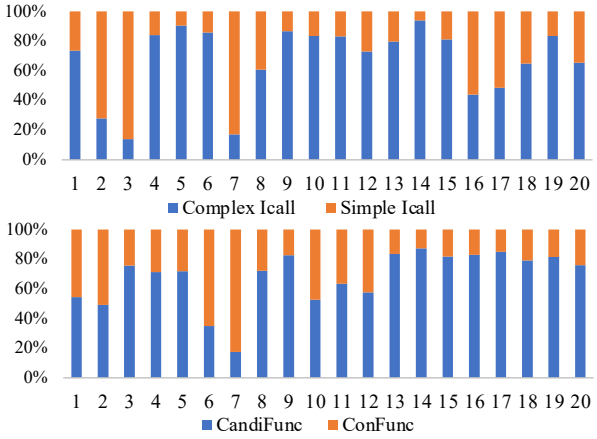


Figure 7: (a) and (b) show the relative proportion of the simple indirect calls (Icall) and the confined functions.

Time Cost. After showing the high precision of KELP, we next assess whether KELP induces negligible time costs. As shown in Table 1, KELP incurs a few extra time costs against MLTA, on average, only increasing by 6.3 seconds and incurring 8.5% time overheads. The negligible time costs are due to the affordable, regional information of simple function pointers, which is efficiently tracked by our def-use tracking. More importantly, KELP remains very scalable and lightweight for the Linux kernel, constructing its precise call graph in around eleven minutes. For each project of the remaining six million-line industrial-strength software programs, KELP can finish its analysis within four minutes. Therefore, considering the average precision improvement of 54.2%, we believe that KELP is promising to serve as an efficient and precise pre-processing step for pure type analysis.

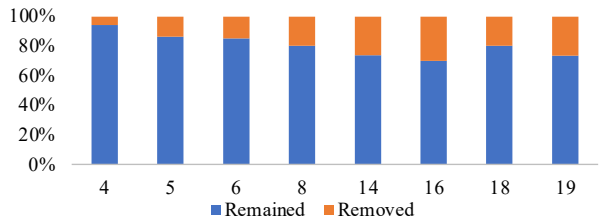


Figure 8: Statements that do not access thread-shared memory in FSAM are removed by using KELP's CGs.

6.2 Enhancing Downstream Clients

In § 6.1, we have shown that KELP advances the state-of-the-art type analysis. Furthermore, we assess the KELP's effectiveness through the lens of three downstream applications: thread-sharing analysis [35, 76, 85], value-flow bug detection [21, 68, 81], and directed grey-box fuzzing [17, 34, 96]. Importantly, these clients' effectiveness strongly relies on the precision of the call graphs being provided (explained in § 6.2.1, § 6.2.2, and § 6.2.3).

Experiment Setup. To select the tools of these downstream clients, first, we chose FSAM [76] in the MTA module of SVF [79] as thread-sharing analysis to identify loads and stores accessing thread-shared memory. Second, we chose SABER [81] in SVF as a bug finding tool for checking source-sink properties. More specifically, we used the three built-in checkers in SABER, the memory leak checker, file leak checker, and double free checker. Note that SVF is a popular tool that underpins many security applications [18–20, 37, 40, 42, 58]. Third, we used the recent work BEACON [34] as our directed fuzzer for the contexts of bug reproduction. In our dedicated evaluation of call graphs produced by KELP and the pure type analysis (the focus of this paper), we constructed call graphs using the two different sets of indirect-call results. Our examination measured the extent of improvement in the effectiveness of each tool when utilizing call graphs of varying precision.

6.2.1 Thread-Sharing Analysis

Thread-sharing analysis (TSA) that determines whether a statement can read or write thread-shared data shores up many popular applications, such as understanding and debugging concurrent programs [36, 53, 73], and concurrency bug detection [7, 18, 37]. By the nature of the inter-procedural analysis, FSAM [76], the TSA tool, requires a precise CG to identify the related statements.

We chose the eight software programs in Table 1 that use Pthread APIs as our benchmarks. To perform TSA for large codebases, we chose Steensgaard's pointer analysis [75] in the WPA module of SVF. We evaluate the reduction in false thread-sharing statements in FSAM by using the CGs generated by KELP and MLTA, respectively. As illustrated in Figure 8, on average, 25.3% of false thread-sharing statements can be additionally removed by using KELP's call graphs. As shown in Table 1, the precision improvement is due to further removing 55.5% of bogus indirect-call targets by KELP, with a few (9.2%) extra time costs in building CGs. To sum up, KELP is effective in helping thread-sharing analysis.

Table 2: Precise call graphs for finding value-flow bugs.

Project	Memory Leak	File Leak	Double Free
(1) OpenSSH	78 (71, -7)	37 (32, -5)	21 (18, -3)
(2) GCC	50 (41, -9)	6 (6, -0)	2 (2, -0)
(3) Curl	6 (6, -0)	14 (11, -3)	5 (5, -0)
(4) redis	124 (94, -30)	64 (58, -6)	42 (39, -3)
(6) zfs	95 (81, -14)	32 (21, -11)	111 (91, -20)
(12) Wrk	281 (223, -58)	74 (61, -13)	250 (211, -39)

6.2.2 Source-Sink Value-Flow Bug Detection

Value-flow analysis [14, 21, 68] is powerful at checking many source-sink properties, such as heap memory errors. SABER [81] detects source-sink value-flow bugs by tracking a global value-flow graph. We used the six software programs from Table 1 that did not run out of memory in our running environment as our benchmarks. Intuitively, more precise CGs for the same tool result in more precise control flows during bug detection, leading to removing false bug warnings.

As shown in Figure 2, KELP can help SABER to further reduce 17.1% of false positives. As shown in Table 1, the precision improvement is due to further removing 54.9% of bogus indirect-call targets by KELP, with only a few (6.2%) extra time costs in building CGs. By examining bug reports, we observed that the less than 20% false-positive reduction is reasonable because many bugs do not span across indirect control flows. In addition, some false warnings in the reports are induced by other factors (e.g., path conditions, imprecise points-to sets of general pointers). A precise and efficient vulnerability hunting tool requires various efforts, and we focus on improving call graphs. To sum up, KELP's CGs are effective in helping value-flow bug detection.

6.2.3 Directed Grey-Box Fuzzing

Directed grey-box fuzzing is a promising technique for bug reproduction, a crucial part of bug understanding and fixing [36, 49]. The recent work, BEACON [34], speeds up the bug reproduction process by pruning away irrelevant paths that cannot reach the target code. To this end, BEACON performs static control-flow reachability and "asserts" irrelevant paths through instrumentation. As a result, the executions irrelevant to the target code are stopped early during fuzzing. At a high level, more

Table 3: Precise CGs for directed grey-box fuzzing (the numbers in parentheses represent the improvement of KELP).

Project	CVE ID	T_{stat} (s)	$T_{fuzzing}$ (h)	T_{all} (h)	#Executions (million)	Avg. Callee Size
Ming-4.7	2016-9827	6.3 (7.0, +0.7)†	0.79 (0.40, 49.2%↓)	0.79 (0.40, 49.2%↓)	1.20 (0.71, 41.0%↓)	5.5 (3.8, 30.9%↓)
	2016-9829	6.3 (7.0, +0.7)	4.09 (1.30, 68.3%↓)	4.10 (1.30, 68.3%↓)	8.88 (1.99, 77.6%↓)	
	2017-11728	6.3 (7.0, +0.7)	1.78 (1.10, 38.2%↓)	1.78 (1.10, 38.2%↓)	2.27 (1.79, 21.2%↓)	
	2017-11729	6.3 (7.0, +0.7)	2.73 (0.98, 64.0% ↓)	2.74 (0.99, 64.0% ↓)	3.92 (1.77, 54.8%↓)	
Ming-4.8	2018-8807	7.2 (8.1, +0.9)	2.88 (1.53, 47.0%↓)	2.88 (1.53, 47.0%↓)	28.57 (9.82, 65.6%↓)	3.1 (1.8, 41.9%↓)
	2018-8962	7.2 (8.1, +0.9)	5.09 (2.44, 52.0%↓)	5.09 (2.44, 52.0%↓)	22.14 (17.19, 22.4%↓)	
libxml	2017-5969	839.0 (846.9, +7.9)	1.05 (0.11, 89.2%↓)	1.28 (0.35, 72.8%↓)	4.02 (0.59, 85.4%↓)	11.9 (3.9, 67.2%↓)
binutils	2017-7209	46.3 (48.1, +1.8)	1.33 (0.57, 57.0%↓)	1.34 (0.58, 56.5%)	6.94 (3.78, 45.5%↓)	2.1(1.8, 14.3%↓)
binutils	2020-16590	83.1 (86.3, +3.2)	4.68 (2.87, 38.7%↓)	4.70 (2.98, 36.5%↓)	42.16 (24.52, 41.8%↓)	4.1 (2.2, 46.4%↓)
binutils	2020-16591	54.2 (57.3, +3.1)	10.55 (6.95, 34.1%↓)	10.57 (6.97, 34.1%↓)	68.11 (58.66, 13.9%↓)	4.1 (2.2, 46.4%↓)

The versions of binutils are 53f7e8ea, f717994, and c98a454 from top to bottom. †: the difference is spent in CG construction.

precise CGs lead to more precise control-flow reachability and, thus, more precise path pruning. Consequently, the fuzzer can explore fewer infeasible program paths, thereby increasing the reproduction efficiency.

We followed the benchmarks [34] and chose ten previous CVE IDs impacted by indirect-call control flows from Libxml, Binutils, and Ming spreading across different historical versions. We assess (i) the time costs (second) of static analyses (e.g., building CGs to perform control flow reachability), denoted as T_{stat}. In addition, we measure (ii) the time costs of reproducing the CVEs through fuzzing $T_{fuzzing}$ and (iii) the number of executions required (million) to reproduce the CVEs #*Executions*. Specifically, more effective CG construction leads to more effective control-flow path pruning, thus resulting in fewer time costs $T_{stat} + T_{fuzzing}$ and fewer executions #*Executions* to reproduce bugs.

Five runs of fuzzing were performed to compute the average results shown in Table 3. On average, KELP can help BEACON further reduce 51.9% of time costs ($T_{stat} + T_{fuzzing}$) and 46.9% of executions (#*Executions*). Specifically, by using our CGs for reproducing the ten CVEs, BEACON only needs to additionally pay a few seconds (fewer than ten seconds for each project) in T_{stat}, thereby reducing 16.618 hours in whole time costs T_{all} and 67.389 millions of executions. Shown in Table 3, the efficiency improvement is due to additionally reducing 49.3% of bogus indirect-call targets by KELP. To sum up, KELP's CGs are effective in helping directed fuzzing.

6.3 Discussion

Constant Propagation. Constant propagation [64] is a technique used in compiler optimization and static analysis to determine and propagate constant values through a program. It could handle single-callee indirect calls by capturing the constant values of function-pointer variables. We perform an experiment that compares KELP to MLTA powered by constant propagation. Our results on the twenty software programs show that KELP is 45.7% more precise than MLTA with constant propagation, with only about 4 seconds of additional overheads. In conclusion, constant-value propagation is ineffective for handling numerous simple indirect calls that can have multiple callees across different calling contexts due to the dynamic nature of function pointers.

Ablation Study. Next, we perform an ablation analysis on each stage of KELP to assess the precision enhancement achieved by characterizing simple function pointers and confined functions. The precision improvement is measured using the average callee size as a metric. Our results show that when only the first stage is used, KELP could only improve MLTA by 32.7%. When only the second stage is used, KELP could only improve MLTA by 23.2%. In conclusion, the combined use of both stages in KELP can result in a significantly higher level of precision compared to MLTA.

False-Negative Analysis. We also investigate whether KELP may incur new false negatives in regional def-use tracking. To reduce subjectivity and collect ground truths, we implemented a trace collection tool using Intel PT-based (Processor Tracing) to dynamically collect indirect-call traces through instrumentation. To alleviate the poor-coverage limitation of dynamic execution, we used AFL++ as our fuzzer to extensively test the three popular software programs (with six different versions shown in Table 3) under various inputs in around one week. Since the traces have debug information (e.g., the line number), we used such information to identify the matched callees. By analyzing the traces, initially, we found a few callees were missed at eight indirect calls, all due to the implicit casting of primitive types in the type analysis (e.g., casting from `long int` to `char *`). The reasons for these false negatives in type analysis are well studied by the previous work [50, 55, 82]. We have fixed these cases by equalizing certain primitive types [55]. Consequently, at the time of writing, we did not find any new false negatives. In another experiment, shown in Table 3, KELP did not remove the real callees in indirect calls leading to the target code and succeeded in reproducing the CVE IDs. In summary, KELP does not induce extra false negatives owing to our safe fallback strategy.

7 Related Work

A long stream of downstream program analysis applications [13–15, 24, 68, 93] require effective CG construction. We discuss two lines of existing CG construction.

Pointer Analysis. Pointer analysis is an essential technique with much seminal work [32, 69]. Particularly, many important dimensions are developed to determine how precisely the pointer information is computed, including field-sensitivity [10, 59], flow-sensitivity [30, 31, 38, 47], context-sensitivity [45, 48, 70], and inclusion-based or unification-based analyses [4, 29, 60, 75]. In principle, more precise pointer analysis can bring more performance overheads and, thus, be less efficient for large programs. Much research may focus on other problems (e.g., bug finding [5, 38, 39, 72], persistent pointer information [67, 86]). We discuss the pointer analysis from the problem of the indirect-call resolution.

The existing pointer analysis could be categorized into exhaustive analysis (function pointers and general pointers are computed simultaneously) and on-demand analysis (function pointers are resolved when needed). On-demand pointer analysis [74, 77, 80] is more efficient for resolving function pointers by reducing redundant computation on general data pointers. However, existing precise demand-driven pointer analysis is still very hard to scale up to hundreds of thousands of lines of code.

More specifically, SUPA [77, 80] is the state-of-the-art field-, flow-, and context-sensitive Andersen-style on-demand pointer analysis. SUPA first performs exhaustive flow-insensitive Andersen's analysis [29] to construct a conservative value-flow graph that captures the def-use relations of variables and objects. Then, for a function pointer of interest, SUPA conducts points-to refinements based on the graph. Despite the advancement, the recent work [40] still shows that SUPA has difficulty in scaling up to GCC (135 KLoC) within twelve hours to resolve function pointers. In contrast to SUPA, KELP differentiates simple function pointers from the complex ones, using the regional pointer information of direct value flows to resolve the simple pointers precisely. In addition, our def-use analysis can make type analysis more precise by reducing the candidate address-taken functions. Therefore, as shown in Table 1, KELP is highly scalable for the twenty programs with more than 100 KLoC.

Lastly, we delve into the differences between KELP and our other concurrently published work, CORAL [16]. Specifically, CORAL defines the concept of stationary function pointers, which can be precisely resolved even with flow-insensitive unification-based or inclusion-based analyses. The results show that CORAL can scale up to millions of lines, analyzing MariaDB (1.8 MLoC) within 2.5 hours. First, it is important to note that stationary function pointers are complex because they can be referenced by other pointers [16]. KELP utilizes scalable type analysis to resolve the indirect calls using stationary and complex function pointers, whereas CORAL adopts a progressive and refinement-based approach, combining multiple pointer analyses to achieve higher precision. Second, CORAL is designed for scenarios where time and memory costs can be tolerated in order to embrace high precision, such as the daily build process [68]. On the other hand, KELP can analyze millions of lines of code in just a few minutes, making it suitable for scenarios that prioritize efficiency, such as the continuous integration process. In summary, KELP and CORAL leverage different characteristics of function pointers, are tailored to different scenarios, and harness different techniques.

Type Analysis. Type analysis can scale up to millions of lines of code in minutes. First, function signature analysis (FSA) [6] identifies indirect-call targets by matching the types of function pointers (used in indirect calls) with those of candidate address-taken functions. Due to the low costs, FSA has been extensively used for control-flow integrity [12, 55, 56, 91, 92, 95]. Some work [26, 46] additionally considers field types of two-layer structs. To improve the precision, multi-layer type analysis [50] considers the field types of multi-layer structures to prune away infeasible indirect-call targets. However, as shown in our experiments and other literature [28, 40, 94, 94], pure type analysis can still induce many false positives. To address the precision problem, KELP uses the regional pointer information of simple pointers for the first time, thereby improving the precision of both complex and simple indirect calls.

Finally, much seminal research shares a different problem scope with us, constructing CGs for binaries [8, 9, 43] or other high-level languages (e.g., Java [1, 2, 62], Python [65], C# [66], JavaScript [54, 71], and Scala [3, 61]), or resorting to dynamic program information [23, 25, 33, 57, 84]. We believe that extending our insights and approaches to other high-level languages is an exciting direction, and we leave it to our future work.

8 Conclusion

We have introduced a staged and concerted approach named KELP to improve the precision of type-based call graph construction with negligible additional time overheads. Our extensive experiments have shown that using regional pointer information for simple function pointers is a big stride forward in the realm of type-based CG construction, significantly improving the effectiveness of refining indirect-call targets. We expect that our work can improve more downstream program analysis clients and also offer intriguing insights for other static analyses.

Acknowledgments

We express our gratitude to the anonymous reviewers for their valuable feedback. This work is supported by the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei.

References

[1] ALI, K., LAI, X., LUO, Z., LHOTÁK, O., DOLBY, J., AND TIP, F. A study of call graph construction for jvm-hosted languages. *IEEE Trans. Software Eng. 47*, 12 (2021), 2644–2666.

[2] ALI, K., AND LHOTÁK, O. Application-only call graph construction. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings* (2012), J. Noble, Ed., vol. 7313 of *Lecture Notes in Computer Science*, Springer, pp. 688–712.

[3] ALI, K., RAPOPORT, M., LHOTÁK, O., DOLBY, J., AND TIP, F. Type-based call graph construction algorithms for scala. *ACM Trans. Softw. Eng. Methodol. 25*, 1 (2015), 9:1–9:43.

[4] ANDERSEN, L. O. Program analysis and specialization for the c programming language. Tech. rep., DIKU, 1994.

[5] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., TRAON, Y. L., OCTEAU, D., AND MCDANIEL, P. D. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (2014), M. F. P. O'Boyle and K. Pingali, Eds., ACM, pp. 259–269.

[6] ATKINSON, D. C. Accurate call graph extraction of programs with function pointers using type signatures. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea* (2004), IEEE Computer Society, pp. 326–335.

[7] BAI, J., LAWALL, J., CHEN, Q., AND HU, S. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), D. Malkhi and D. Tsafrir, Eds., USENIX Association, pp. 255–268.

[8] BALAKRISHNAN, G., GRUIAN, R., REPS, T. W., AND TEITELBAUM, T. Codesurfer/x86-a platform for analyzing x86 executables. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings* (2005), R. Bodík, Ed., vol. 3443 of *Lecture Notes in Computer Science*, Springer, pp. 250–254.

[9] BALAKRISHNAN, G., AND REPS, T. W. Analyzing memory accesses in x86 executables. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings* (2004), E. Duesterwald, Ed., vol. 2985 of *Lecture Notes in Computer Science*, Springer, pp. 5–23.

[10] BALATSOURAS, G., AND SMARAGDAKIS, Y. Structure-sensitive points-to analysis for C and C++. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings* (2016), X. Rival, Ed., vol. 9837 of *Lecture Notes in Computer Science*, Springer, pp. 84–104.

[11] BÖHME, M., PHAM, V., NGUYEN, M., AND ROYCHOUDHURY, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, pp. 2329–2344.

[12] BUROW, N., CARR, S. A., NASH, J., LARSEN, P., FRANZ, M., BRUNTHALER, S., AND PAYER, M. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv. 50*, 1 (2017), 16:1–16:33.

[13] CAI, Y., YAO, P., YE, C., AND ZHANG, C. Place your locks well: Understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023* (2023), J. A. Calandrino and C. Troncoso, Eds., USENIX Association.

[14] CAI, Y., YAO, P., AND ZHANG, C. Canary: practical static detection of inter-thread value-flow bugs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2021), S. N. Freund and E. Yahav, Eds., ACM, pp. 1126–1140.

[15] CAI, Y., YE, C., SHI, Q., AND ZHANG, C. Peahen: fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022* (2022), A. Roychoudhury, C. Cadar, and M. Kim, Eds., ACM, pp. 784–796.

[16] CAI, Y., AND ZHANG, C. A cocktail approach to practical call graph construction. *Proc. ACM Program. Lang. 7*, OOPSLA2 (2023).

[17] CANAKCI, S., MATYUNIN, N., GRAFFI, K., JOSHI, A., AND EGELE, M. Targetfuzz: Using darts to guide directed greybox fuzzers. In *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022* (2022), Y. Suga, K. Sakurai, X. Ding, and K. Sako, Eds., ACM, pp. 561–573.

[18] CHEN, H., GUO, S., XUE, Y., SUI, Y., ZHANG, C., LI, Y., WANG, H., AND LIU, Y. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (2020), S. Capkun and F. Roesner, Eds., USENIX Association, pp. 2325–2342.

[19] CHEN, H., XUE, Y., LI, Y., CHEN, B., XIE, X., WU, X., AND LIU, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, pp. 2095–2108.

[20] CHEN, Y., LI, P., XU, J., GUO, S., ZHOU, R., ZHANG, Y., WEI, T., AND LU, L. SAVIOR: towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020* (2020), IEEE, pp. 1580–1596.

[21] CHEREM, S., PRINCEHOUSE, L., AND RUGINA, R. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 480–491.

[22] CORINA, J., MACHIRY, A., SALLS, C., SHOSHITAISHVILI, Y., HAO, S., KRUEGEL, C., AND VIGNA, G. DIFUZE: interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS* (2017), B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, pp. 2123–2138.

[23] DING, R., QIAN, C., SONG, C., HARRIS, W., KIM, T., AND LEE, W. Efficient protection of path-sensitive control security. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017* (2017), E. Kirda and T. Ristenpart, Eds., USENIX Association, pp. 131–148.

[24] FAN, G., WU, R., SHI, Q., XIAO, X., ZHOU, J., AND ZHANG, C. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019* (2019), J. M. Atlee, T. Bultan, and J. Whittle, Eds., IEEE / ACM, pp. 72–82.

[25] GE, X., CUI, W., AND JAEGER, T. GRIFFIN: guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017* (2017), Y. Chen, O. Temam, and J. Carter, Eds., ACM, pp. 585–598.

[26] GE, X., TALELE, N., PAYER, M., AND JAEGER, T. Fine-grained control-flow integrity for kernel software. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016* (2016), IEEE, pp. 179–194.

[27] GENS, D., SCHMITT, S., DAVI, L., AND SADEGHI, A. K-miner: Uncovering memory corruption in linux. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (2018), The Internet Society.

[28] GHAVAMNIA, S., PALIT, T., MISHRA, S., AND POLYCHRONAKIS, M. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (2020), S. Capkun and F. Roesner, Eds., USENIX Association, pp. 1749–1766.

[29] HARDEKOPF, B., AND LIN, C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 290–299.

[30] HARDEKOPF, B., AND LIN, C. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009* (2009), Z. Shao and B. C. Pierce, Eds., ACM, pp. 226–238.

[31] HARDEKOPF, B., AND LIN, C. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011* (2011), IEEE Computer Society, pp. 289–298.

[32] HIND, M. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001* (2001), J. Field and G. Snelting, Eds., ACM, pp. 54–61.

[33] HU, H., QIAN, C., YAGEMANN, C., CHUNG, S. P. H., HARRIS, W. R., KIM, T., AND LEE, W. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, pp. 1470–1486.

[34] HUANG, H., GUO, Y., SHI, Q., YAO, P., WU, R., AND ZHANG, C. BEACON: directed grey-box fuzzing with provable path pruning. In *43rd IEEE Symposium on Security and Privacy, SP 2022* (2022), IEEE, pp. 36–50.

[35] HUANG, J. Scalable thread sharing analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (2016), L. K. Dillon, W. Visser, and L. A. Williams, Eds., ACM, pp. 1097–1108.

[36] HUANG, J., ZHANG, C., AND DOLBY, J. CLAP: recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013* (2013), H. Boehm and C. Flanagan, Eds., ACM, pp. 141–152.

[37] JEONG, D. R., KIM, K., SHIVAKUMAR, B., LEE, B., AND SHIN, I. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019* (2019), IEEE, pp. 754–768.

[38] KAHLON, V. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, Association for Computing Machinery, pp. 249 – 259.

[39] KAHLON, V., YANG, Y., SANKARANARAYANAN, S., AND GUPTA, A. Fast and accurate static data-race detection for concurrent programs. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings* (2007), W. Damm and H. Hermanns, Eds., vol. 4590 of *Lecture Notes in Computer Science*, Springer, pp. 226–239.

[40] KHANDAKER, M., LIU, W., NASER, A., WANG, Z., AND YANG, J. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019* (2019), N. Heninger and P. Traynor, Eds., USENIX Association, pp. 195–211.

[41] KHANDAKER, M., NASER, A., LIU, W., WANG, Z., ZHOU, Y., AND CHENG, Y. Adaptive call-site sensitive control flow integrity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019* (2019), IEEE, pp. 95–110.

[42] KIM, C. H., KIM, T., CHOI, H., GU, Z., LEE, B., ZHANG, X., AND XU, D. Securing real-time microcontroller systems through customized memory view switching. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (2018), The Internet Society.

[43] KIM, S. H., SUN, C., ZENG, D., AND TAN, G. Refining indirect call targets at the binary level. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021* (2021), The Internet Society.

[44] KIM, T., AND ZELDOVICH, N. Making linux protection mechanisms egalitarian with userfs. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings* (2010), USENIX Association, pp. 13–28.

[45] LATTNER, C., LENHARTH, A., AND ADVE, V. S. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 278–289.

[46] LI, J., TONG, X., ZHANG, F., AND MA, J. Fine-cfi: Fine-grained control-flow integrity for operating system kernels. *IEEE Trans. Inf. Forensics Secur. 13*, 6 (2018), 1535–1550.

[47] LI, L., CIFUENTES, C., AND KEYNES, N. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (New York, NY, USA, 2011), ESEC/FSE '11, Association for Computing Machinery, pp. 343–353.

[48] LI, Y., TAN, T., MØLLER, A., AND SMARAGDAKIS, Y. A principled approach to selective context sensitivity for pointer analysis. *ACM Trans. Program. Lang. Syst. 42*, 2 (2020), 10:1–10:40.

[49] LIU, H., SILVESTRO, S., WANG, W., TIAN, C., AND LIU, T. ireplayer: in-situ and identical record-and-replay for multithreaded applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018* (2018), J. S. Foster and D. Grossman, Eds., ACM, pp. 344–358.

[50] LU, K., AND HU, H. Where does it go?: Refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019* (2019), L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds., ACM, pp. 1867–1881.

[51] LYU, Y., FANG, Y., ZHANG, Y., SUN, Q., MA, S., BERTINO, E., LU, K., AND LI, J. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022* (2022), IEEE, pp. 2096–2113.

[52] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017* (2017), E. Kirda and T. Ristenpart, Eds., USENIX Association, pp. 1007–1024.

[53] NANDA, M. G., AND RAMESH, S. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst. 28*, 6 (nov 2006), 1088–1144.

[54] NIELSEN, B. B., TORP, M. T., AND MØLLER, A. Modular call graph construction for security scanning of node.js applications. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021* (2021), C. Cadar and X. Zhang, Eds., ACM, pp. 29–41.

[55] NIU, B., AND TAN, G. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (2014), M. F. P. O'Boyle and K. Pingali, Eds., ACM, pp. 577–587.

[56] NIU, B., AND TAN, G. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014* (2014), G. Ahn, M. Yung, and N. Li, Eds., ACM, pp. 1317–1328.

[57] NIU, B., AND TAN, G. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015* (2015), I. Ray, N. Li, and C. Kruegel, Eds., ACM, pp. 914–926.

[58] ORENBACH, M., MICHALEVSKY, Y., FETZER, C., AND SILBERSTEIN, M. Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), D. Malkhi and D. Tsafrir, Eds., USENIX Association, pp. 555–570.

[59] PEARCE, D. J., KELLY, P. H., AND HANKIN, C. Efficient field-sensitive pointer analysis of c.

[60] PEREIRA, F. M. Q., AND BERLIN, D. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (USA, 2009), CGO '09, IEEE Computer Society, pp. 126 – 135.

[61] PETRASHKO, D., URECHE, V., LHOTÁK, O., AND ODERSKY, M. Call graphs for languages with parametric polymorphism. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016* (2016), E. Visser and Y. Smaragdakis, Eds., ACM, pp. 394–409.

[62] REIF, M., EICHBERG, M., HERMANN, B., LERCH, J., AND MEZINI, M. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, pp. 474 – 486.

[63] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), POPL '95, Association for Computing Machinery, p. 49–61.

[64] SAGIV, S., REPS, T. W., AND HORWITZ, S. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci. 167*, 1&2 (1996), 131–170.

[65] SALIS, V., SOTIROPOULOS, T., LOURIDAS, P., SPINELLIS, D., AND MITROPOULOS, D. Pycg: Practical call graph generation in python. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021* (2021), IEEE, pp. 1646–1657.

[66] SANTHIAR, A., AND KANADE, A. Static deadlock detection for asynchronous c# programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (2017), A. Cohen and M. T. Vechev, Eds., ACM, pp. 292–305.

[67] SCHUBERT, P., HERMANN, B., AND BODDEN, E. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis. *European Conference on Object-Oriented Programming (ECOOP)* (2021).

[68] SHI, Q., WU, R., FAN, G., AND ZHANG, C. Conquering the extensional scalability problem for value-flow analysis frameworks. *CoRR abs/1912.06878* (2019).

[69] SMARAGDAKIS, Y., AND BALATSOURAS, G. Pointer analysis. *Found. Trends Program. Lang. 2*, 1 (2015), 1–69.

[70] SMARAGDAKIS, Y., KASTRINIS, G., AND BALATSOURAS, G. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (2014), M. F. P. O'Boyle and K. Pingali, Eds., ACM, pp. 485–495.

[71] SOTIROPOULOS, T., AND LIVSHITS, B. Static analysis for asynchronous javascript programs. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom* (2019), A. F. Donaldson, Ed., vol. 134 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 8:1–8:30.

[72] SPÄTH, J., DO, L. N. Q., ALI, K., AND BODDEN, E. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy* (2016), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 22:1–22:26.

[73] SRIDHARAN, M., FINK, S. J., AND BODÍK, R. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 112–122.

[74] SRIDHARAN, M., GOPAN, D., SHAN, L., AND BODÍK, R. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA* (2005), R. E. Johnson and R. P. Gabriel, Eds., ACM, pp. 59–76.

[75] STEENSGAARD, B. Points-to analysis in almost linear time. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996* (1996), H. Boehm and G. L. S. Jr., Eds., ACM Press, pp. 32–41.

[76] SUI, Y., DI, P., AND XUE, J. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016* (2016), B. Franke, Y. Wu, and F. Rastello, Eds., ACM, pp. 160–170.

[77] SUI, Y., AND XUE, J. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016* (2016), T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds., ACM, pp. 460–473.

[78] SUI, Y., AND XUE, J. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction* (2016), ACM, pp. 265–266.

[79] SUI, Y., AND XUE, J. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016* (2016), A. Zaks and M. V. Hermenegildo, Eds., ACM, pp. 265–266.

[80] SUI, Y., AND XUE, J. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Trans. Software Eng. 46*, 8 (2020), 812–835.

[81] SUI, Y., YE, D., AND XUE, J. Static memory leak detection using full-sparse value-flow analysis. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012* (2012), M. P. E. Heimdahl and Z. Su, Eds., ACM, pp. 254–264.

[82] TAN, G., AND JAEGER, T. CFG construction soundness in control-flow integrity. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017* (2017), ACM, pp. 3–13.

[83] THE LLVM PROJECT. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register, 2021. Accessed on October 2, 2023.

[84] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAS, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015* (2015), I. Ray, N. Li, and C. Kruegel, Eds., ACM, pp. 927–940.

[85] WHALEY, J., AND RINARD, M. C. Compositional pointer and escape analysis for java programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1999, Denver, Colorado, USA, November 1-5, 1999* (1999), B. Hailpern, L. M. Northrop, and A. M. Berman, Eds., ACM, pp. 187–206.

[86] XIAO, X., ZHANG, Q., ZHOU, J., AND ZHANG, C. Persistent pointer information. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (2014), M. F. P. O'Boyle and K. Pingali, Eds., ACM, pp. 463–474.

[87] XU, M., KASHYAP, S., ZHAO, H., AND KIM, T. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020* (2020), IEEE, pp. 1643–1660.

[88] XU, M., QIAN, C., LU, K., BACKES, M., AND KIM, T. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA* (2018), IEEE Computer Society, pp. 661–678.

[89] YAN, H., SUI, Y., CHEN, S., AND XUE, J. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (2018), M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds., ACM, pp. 327–337.

[90] YOO, S., PARK, J., KIM, S., KIM, Y., AND KIM, T. In-kernel control-flow integrity on commodity oses using ARM pointer authentication. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022* (2022), K. R. B. Butler and K. Thomas, Eds., USENIX Association, pp. 89–106.

[91] ZENG, B., TAN, G., AND MORRISETT, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011* (2011), Y. Chen, G. Danezis, and V. Shmatikov, Eds., ACM, pp. 29–40.

[92] ZENG, D., NIU, B., AND TAN, G. Mazerunner: Evaluating the attack surface of control-flow integrity policies. In *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021* (2021), IEEE, pp. 810–821.

[93] ZHANG, H., CHEN, W., HAO, Y., LI, G., ZHAI, Y., ZOU, X., AND QIAN, Z. Statically discovering high-order taint style vulnerabilities in OS kernels. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021* (2021), Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., ACM, pp. 811–824.

[94] ZHANG, T., SHEN, W., LEE, D., JUNG, C., AZAB, A. M., AND WANG, R. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019* (2019), N. Heninger and P. Traynor, Eds., USENIX Association, pp. 1205–1220.

[95] ZHANG, Y., LIU, X., SUN, C., ZENG, D., TAN, G., KAN, X., AND MA, S. Recfa: Resilient control-flow attestation. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021* (2021), ACM, pp. 311–322.

[96] ZONG, P., LV, T., WANG, D., DENG, Z., LIANG, R., AND CHEN, K. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (2020), S. Capkun and F. Roesner, Eds., USENIX Association, pp. 2255–2269.