A Hybrid Alias Analysis and Its Application to Global Variable Protection in the Linux Kernel

Guoren Li , Hang Zhang[*] , Jinmeng Zhou[†] , Wenbo Shen[†] , Yulei Sui[‡] , and Zhiyun Qian

University of California, Riverside [*]Georgia Institute of Technology
[†]Zhejiang University [‡]University of New South Wales

Abstract

Global variables in the Linux kernel have been a common target of memory corruption attacks to achieve privilege escalation. Several potential defense mechanisms can be employed to safeguard global variables. One approach involves placing global variables in read-only pages after kernel initialization (`ro_after_init`), while another involves employing software fault isolation (SFI) to dynamically block unintended writes to these variables. To deploy such solutions in practice, a key building block is a sound, precise, and scalable alias analysis that is capable of identifying all the pointer aliases of global variables, as any pointer alias may be used for intended writes to a global variable. Unfortunately, the two existing styles of data-flow-based (*e.g.,* Andersen-style) alias analysis and type-based alias analysis have serious limitations in scalability and precision when applied to the Linux kernel.

This paper proposes a novel and general hybrid alias analysis that unifies the two complementary approaches in a graph reachability framework using context-free-language, also known as CFL-reachability. We show our hybrid alias analysis is extremely effective, significantly and simultaneously outperforming the data-flow-based alias analysis in scalability and the type-based alias analysis in precision. Under the same time budget, our hybrid analysis finds 42% of the Linux kernel global variables protectable as `ro_after_init`, whereas the two separate analyses find a combined 16% only.

1 Introduction

Kernel exploits of memory corruption vulnerabilities have evolved over time together with defenses. Today exploits generally target important kernel data in the form of either control or non-control data. Control data can be function pointers and return addresses whereas non-control data can be various variables that have specific OS semantics [58] (`uid` [13, 33, 44], `modprobe_path` [24, 53, 66], `selinux_enforcing` [7, 27]).

Global data are common targets for two reasons among all kernel data. First, the addresses of global data are determined at compile time and usually have fixed runtime addresses. Kernel Address Space Layout Randomization (KASLR) will randomize the base address of all global data, but it is unfortunately easy to bypass because of the prevalence of kernel information leak vulnerabilities [14] and recent architectural side-channel attacks [18, 22, 23, 29, 31].

Second, many valuable memory corruption targets exist in global data, including both control and non-control data. These global variables are in read/write data sections, which can be easily corrupted by memory corruption bugs. For example, function pointers `fsync` and `check_flags` within `ptmx_fops` have been commonly exploited in real-world exploits [6, 7]. As examples of non-control data, `modprobe_path` is commonly corrupted by real-world exploits to execute an attacker-specified executable as a root user [24, 53, 66]; `selinux_enforcing` is corrupted to disable SELinux [7].

To minimize exploitable global variables, a few candidate solutions are available, *e.g.,* marking global variables as read-only (see §2 for details) when appropriate. However, to utilize them widely in the Linux kernel, one needs to generate a sound security policy automatically, necessitating a sound alias analysis — determining which pointers can point to a given global variable. This is non-trivial because a global variable can be accessed indirectly through pointers that are propagated throughout the kernel via complex procedures involving multiple syscalls. Failing to find pointer aliases to global variables can result in intended writes being misclassified as unintended, leading to runtime errors.

In the literature, two styles of alias analysis exist: (1) data-flow-based alias analysis (*e.g.,* Andersen-style [10]), and (2) type-based alias analysis [17, 37]. However, while precise, the former suffers from scalability issues when analyzing programs as complex as the Linux kernel. On the other hand, the latter is much more scalable but suffers from significant precision losses.

In this paper, we propose a novel and general hybrid alias analysis that unifies the data-flow-based and type-based alias analysis. Similar to a branch of data-flow-based alias analysis [49, 65], we formulate the problem as a graph reachability

problem, following a context-free-language, also known as CFL-reachability problem [46]. However, rather than confining ourselves to a singular analysis approach (e.g., data-flow-based) and hoping it is one-size-fits-all, we propose a tailored approach that treats each global variable as distinct. By integrating complementary analysis approaches, we can harness their respective strengths while mitigating their inherent limitations. This allows for flexible tuning of precision and scalability on a per-object basis, more easily meeting specific requirements. Applying our hybrid analysis to the problem of global variable protection, we find about 42% of the global variables protectable by `ro_after_init`. In contrast, the result is only 16% using the combined results of the two approaches applied separately, under the same time budget.

In summary, we make the following contributions:

- **Breakthrough:** We design and formalize a novel, tunable, and general alias analysis that unifies the data-flow-based and type-based alias analysis, for improved scalability and precision, setting a new standard in alias analysis.
- **Impact:** The hybrid alias analysis is suitable for a wide range of applications in the Linux kernel where it is necessary to track the pointer aliases of global or heap objects across the entire kernel in a sound manner.
- **Application**: Our solution is highly effective, finding not only 42% of global variables protectable by `ro_after_init` but also all recently-exploited global variables protectable via either `ro_after_init` or software fault isolation.

2 Background and Motivation

In this section, we elaborate on the significance of global variables and the mechanisms available to safeguard them.

Global variables as exploit targets. Unlike dynamically allocated objects on heap and stack, global variables are located at fixed locations determined statically after compilation (barring KASLR). The Linux kernel uses global variables to maintain various states of the whole system. Some global variables are considered switches, which can turn on and off certain critical features. For example, `selinux_enforcing` is used to turn on/off the entire SELinux system. Also, there are many function pointers in global variables, which have important implications as they can lead to control flow hijacking.

These global variables are desirable to attackers due to the relative ease of locating them. As long as KASLR is bypassed (*e.g.,* through an info leak), an exploit with an arbitrary write primitive can corrupt global variables straightforwardly. As an example, in Blackhat 2017, security researchers demonstrated a root exploit against the security-enhanced Samsung Knox kernel by using 3 global variables, `ptmx_fops`, `poweroff_cmd`, `ss_initialized`, even when the kernel code injection defenses are enabled and kernel credentials are marked as read-only in the kernel and protected by hypervisor [45]. More recently, the global variable `modprobe_path` is frequently leveraged to achieve privilege escalation [12,54].

```
1 static struct fops ptmx_fops __ro_after_init;
2 void tty_default_fops(struct fops *fops) {
3     *fops = tty_fops;
4 }
5 static void __init unix98_pty_init(void){
6     tty_default_fops(&ptmx_fops); // initialize
7 }
```

Figure 1: ro_after_init example in C

Opportunities to protect global variables. Currently, there are two main candidate solutions to protect global variables.

First, it is possible to statically mark global variables as `const` (read-only throughout its lifetime) or `ro_after_init` (read-only after kernel initialization) [2]. In both cases, after the kernel is initialized, the global variable will be placed in read-only memory pages, which are generally difficult to alter [16, 64]). For instance, the global variable `ptmx_fops`, which was previously exploited, is now designated as `ro_after_init` (refer to Figure 1). This variable is initialized exclusively in `unix98_pty_init` and remains unmodified thereafter. In response to a Samsung Knox exploit [7] that targeted `ptmx_fops`, kernel developers have subsequently designated it as `ro_after_init`. However, it is unclear whether there are other similarly protectable global variables.

Second, by leveraging software fault isolation (SFI) mechanisms [9,36,47], we can allow write accesses to a global variable from only "authorized" pointers, *i.e.,* those that intend to write to the variable. Any other write accesses, including those made through erroneously directed (e.g., corrupted) pointers, should be considered unauthorized and prevented. For global variables that are indeed modified after initialization, SFI is a promising fine-grained extension to protect them by limiting write instructions to those intended ones. However, it is unclear which instructions are intended for a given global variable, as the writes can occur through pointers.

Limitations of existing mechanisms. The solutions above require developers to track the reachability of global variables through the Linux kernel. Due to the sheer scale, it is extremely challenging to systematically analyze every single global variable manually, given that there are tens of thousands of them. More specifically, global variables may be referenced indirectly through pointer aliases which may propagate throughout the kernel (even across multiple syscalls). Any of the pointer aliases may be used to perform either read access or write access to a global variable. Out of tens of thousands of global variables in Linux kernel v5.14, we find four thousand global variables manually marked as `const` and only 531 global variables manually marked as `ro_after_init`. We suspect the latter number is substantially lower than the actual number in reality, indicating the lack of a robust and automated solution to track pointer aliases of global variables.

Assumptions. We assume an attacker can successfully corrupt the heap or stack memory of the Linux kernel, *e.g.,* via an out-

```c
struct G{ int* fd; };
int gv1;
struct G gv2;
op_on_int(int* num_ptr){
  *num_ptr = 0; // potentially modify gv1
}
syscall1(){
  gv2.fd = &gv1; // gv1 address taken
}
syscall2(){
  op_on_int(gv2.fd);
}
```

Figure 2: challenge example in C

of-bound write and a use-after-free write. However, when they leverage such vulnerabilities to perform an arbitrary write to a protected global variable, we assume the protections offered by read-only pages or SFI-based solutions are not bypassable.

3 Overview

In this section, we describe the high-level goals and challenges of an automated solution to protect global variables. To this end, we develop a novel alias analysis technique packaged in a static analysis called Unias.

3.1 Goals

The goal of Unias is to identify pointer aliases of global variables and their write dereferences. This enables precise tracking of the pointers that can be utilized to modify specific global variables and identifies the exact program points where such modifications can occur. This is helpful in determining how to protect a global variable: (1) whether a global variable should be classified as `ro_after_init`, *i.e.,* if no write dereferences occur after kernel initialization, or (2) which write instructions should be authorized by SFI. Therefore, the key problem we address in Unias is to identify all pointer aliases that may reference a given global variable. It is crucial to note that such an alias analysis must be sound. This is because failing to identify any alias used to write to a global variable can result in an error or a crash. On the other hand, the alias analysis needs to be precise enough to be useful – protecting as many global variables as possible and as tightly as possible. Finally, the kernel is extremely large and the analysis needs to be scalable.

3.2 Challenges

We use a simplified example in Figure 2 to illustrate the complexity and difficulty of alias analyses for global variables in the Linux kernel. There are 2 global variables `gv1`, `gv2` and 2 syscalls `syscall1()`, `syscall2()`. Specifically, `syscall1()` simply assigns the address of `gv1` to `gv2.fd`. On the other hand, `syscall2()` passes `gv2.fd` (a data pointer) to another function `op_on_int()` which writes a zero to the address.

Multi-entry and soundness challenge. OS kernels are stateful by design where user-space programs can interact with them through syscall entries. User-space programs can invoke syscalls in arbitrary sequences. For example, if `syscall1()` is executed before `syscall2()`, we can see that `gv1` will be modified since the address of `gv1` is "leaked" at line 10 and its value is modified at line 7. On the other hand, if `syscall1()` is not invoked before `syscall2()`, then `gv1` may not be modified.

The Linux kernel comprises hundreds of syscalls, and their possible sequences (*e.g.,* permutations) are infinite. Achieving a precise analysis of these infinite sequences is infeasible. The multi-entry challenge highlights the difficulty of analyzing the Linux kernel, as we must make sound approximations.

Not all alias analysis methods prioritize soundness, particularly when analyzing the Linux kernel for bug detection. Many alias analysis approaches [20, 35, 39, 62] in the Linux kernel use heuristic strategies that sacrifice soundness, as soundness is desirable but not always necessary, depending on the applications. However, Unias must be sound by design, as a false negative (*i.e.,* missing a pointer alias) can lead to falsely protecting a global variable, resulting in a memory error when it attempts to modify a protected (read-only) global variable. Therefore, it is best to start with classic and well-reasoned alias analysis algorithms that have been proven sound, such as Andersen-style alias analysis [46].

Limitation of state-of-art solutions in precision and scalability. General data-flow-based approaches such as Andersen-style [10] and Steensgaard-style [51] are widely used for achieving sound and precise alias analysis. However, neither is sufficient to handle global variables in the Linux kernel. This is because, unlike local variables that have a limited scope and lifetime, global variables are potentially accessible anywhere in the Linux kernel. As a result, the scope of analysis becomes extensive and unscalable for Andersen-style analysis, while Steensgaard-style analysis will imprecisely over-approximate the alias results under such a large scope (since Steensgaard-style analysis sacrifices precision for improved scalability by design). As shown in Figure 2, the def-use chain from where `gv1`'s address is taken to where its value is modified can be complex, potentially involving multiple syscalls and thousands of intermediate variables (including other global variables).

Type-based solutions are known as an alternative to data-flow-based alias analysis [17, 30, 37]. The basic idea is to consider pointers that share compatible types to be potential aliases. For example, pointers pointing to the same struct type are considered aliases. However, type-based solutions are too limited in the context of global variables in the Linux kernel. This is because many global variables are defined as primitive types, *e.g.,* `int`. Applying a type-based solution will cause too many pointer aliases, most of which are false positives. Furthermore, `struct` global variables can still have fields of primitive types that are referenced by address. For example, a field can be address-taken and accessed through a pointer later

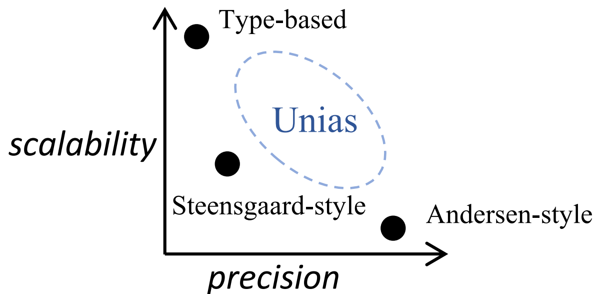

Figure 3: Precision and Scalability Tradeoff Space

on, independent of its base `struct` type. As a result, applying the type-based solution naively can only work for a limited set of global variables, as shown in §6.1.

3.3 Insights

In summary, although both approaches are sound by design, the data-flow-based approach has better precision but worse scalability, while the type-based approach is the opposite. In other words, there is a tradeoff between data-flow-based and type-based approaches in terms of precision and scalability.

Indeed, by evaluating these two approaches individually (see §6), their results are far from satisfactory. We hypothesize that there is a new design space in which we can achieve a much better tradeoff by principally combining the advantages of both approaches, as shown in Figure 3. The idea is to move away from a singular analysis approach and the reliance on a one-size-fits-all solution, we propose a tailored approach that treats each global variable as distinct. Through the integration of complementary approaches, we can leverage their respective strengths while mitigating their inherent limitations, i.e., simultaneously achieving precision close to the Andersen-style approach and scalability close to the type-based approach. This allows for flexible tuning of precision and scalability on a per-variable basis, more easily satisfying specific requirements.

4 Pointer-To-Global Analysis

In this section, we introduce details about the novel design of our pointer-to-global alias analysis that is sound, precise, and scalable. In §4.1, we provide foundational knowledge and terminology that will aid in a better understanding of our design. In §4.2, we introduce our demand-driven alias analysis for global variables based on a CFL graph reachability framework. In §4.3, we integrate type-based shortcuts into the framework to complement the demand-driven analysis, in a tunable fashion, which significantly improves the scalability. In §4.3.3, We give the soundness proof of our design.

4.1 Foundation

Program Assignment Graph (PAG) is a widely used abstract and directed representation of a program that is specifically designed for alias and pointer analysis [4,52], and it is generated from parsing LLVM IR [3]. It is used to represent the pointer manipulation in a given program and capture the points-to relationship.

By design, each node in the PAG represents a variable in the static single assignment (SSA) form. These variables can be memory objects or pointers. Each edge in PAG represents an instruction that manipulates its corresponding nodes and is directional, representing the direction of data flow. In the scope of our analysis, there are five types of edges in PAG: *Addr* (Address-Taken), *Assign*, *Load*, and *Store* are four critical edges that represent a program, and $Field_{t,i,o}$ edge is for field-sensitive analysis, which represents field access in an object, where t refers to the base object type, i refers to the field index, and o refers to the relative byte offset which will be further detailed in §4.2.1. We use $\overline{Assign}$ to represent a reverse edge of *Assign*. A $base \xrightarrow{Field_{t,i,o}} field$ edge represents the pointer arithmetic during struct field accessing, where *base* is the source pointer (of type `t*`) that points to the base of a struct (of type `t`) and *field* is the destination pointer that points to the i_{th} field in the struct. PAG also includes additional information for better analysis, such as variable (node) type, field byte offset, etc.

Since PAG is essentially a form of a data-flow graph capturing relationships between pointers and variables, it is worth noting that PAG retains limited control-flow information (no ordering of edges). It also does not retain the notion of functions. Instead, the graph simply connects actual arguments (at function call sites) to formal arguments (in function definitions) with *Assign* edges; conversely, output variables (*e.g.,* return value from a callee) are connected to the corresponding variable at the caller's callsite. As a result, PAG-based static analysis is by default flow- and context-insensitive.

Alias analysis via CFL-reachability. Andersen-style alias analysis has been formulated as context-free-language (CFL) reachability problems [49,65]. It is essentially a form of graph reachability in which legal paths must be labeled with a string in a specified context-free language; more specifically, in the context of PAG where each edge is labeled (*e.g., Assign*), a legal path would need to have a sequence of edge types that match the language we design. To find whether a pointer may point to a given object (*e.g.,* a global variable) via CFL-reachability, the idea is to check if there exists a path between the pointer node and the object node, such that the path's label is in the CFL which ensures the corresponding program statements (edges) can cause the pointer to point to the object.

Benefits of CFL reachability. Compared to a whole-program alias analysis [10,37], it is worth noting that the CFL reachability problem formulation lends itself naturally to a per-object alias analysis. We leverage this advantage to offer a

tailored analysis approach for each object.

4.2 Demand-Driven Graph Traversal

In this section, we introduce a demand-driven alias analysis naturally formulated as a CFL-reachability problem to identify pointer aliases of global variables through graph traversal on top of PAG (§4.1). In particular, we follow a progressive design: first introducing the initial rules, and then the enhanced rules to model more complex pointer manipulations.

Sensitivity of analysis. As mentioned earlier, analyses built on top of PAG are by default flow- and context- insensitive, which is, in fact, desirable in our problem. Recall the multi-entry and soundness challenge described in §3.2, which requires a flow-insensitive alias analysis that achieves a conservative and sound approximation of infinitely possible sequences of syscall invocations. Besides, being context-sensitive will significantly bloat the graph as multiple copies of callees will be created. Finally, our analysis still attempts to be field-sensitive.

4.2.1 Initial Rules

Given a context-free language L_R (which consists of a set of productions), a PAG G of a C Program P, and a global variable node gv, if gv's address can flow to variable var (*e.g.,* a pointer), through L_R during the execution of P, then var is considered "L_R reachable" from gv in G. The pointer alias set of a global variable `gv` is defined as $pas(gv) = \{n \in G \wedge n \in L_R(gv)\}$, where every node n in $pas(gv)$ is a pointer that "may" point to gv. Hence, determining the pointer aliases of gv is equivalent to computing L_R reachable nodes from gv.

We define the grammar of L_R with the following initial set of productions:

$$\mathbf{F} \rightarrow (\ Assign \mid Store\ \textbf{I-Alias}\ Load\ )* \quad (1)$$

$$\overline{\mathbf{F}} \rightarrow (\ \overline{Assign} \mid \overline{Load}\ \textbf{I-Alias}\ \overline{Store}\ )* \quad (2)$$

$$\textbf{I-Alias} \rightarrow \overline{\mathbf{F}}\ \textbf{I-Alias}\ \mathbf{F} \quad (3)$$

$$\textbf{I-Alias} \rightarrow \overline{Load}\ \textbf{I-Alias}\ Load \quad (4)$$

$$\textbf{I-Alias} \rightarrow \varepsilon \quad (5)$$

$$\textbf{I-Alias} \rightarrow \overline{Field_{t_1,i_1,o}}\ \textbf{I-Alias}\ Field_{t_2,i_2,o} \quad (6)$$

The first production **F** refers to the flows-to relationship that determines which nodes are reachable from a given source node. In LLVM, every global variable is initially accessed through its address in the form of `@gv` (equivalent to `&gv` in C), which is translated to a node a_{gv} in PAG that represents its address. To determine whether a node p is a pointer alias of gv, we evaluate whether a_{gv} can reach p through **F**; if so, we write $\mathbf{F}(a_{gv}, p)$. Production **F** basically looks for sequences of assignments or intermediate memory aliases (*e.g.,* through store and load). For example, $a_{gv} \xrightarrow{Assign} n_1 \xrightarrow{Assign} n_2$ can derive $\mathbf{F}(a_{gv}, n_1)$ and $\mathbf{F}(a_{gv}, n_2)$, as n_1 and n_2 may point to `gv`.

```
1 int gv;
2 struct T1{ char[4] f0; int* f1; };
3 struct T2{ int f2; int* f3; };
4 void func(struct T1* b;){
5     b->f1 = &gv; // gv address taken
6     struct T1* b1 = b;
7     struct T2* b2 = (struct T2*) b1; // cast
8     int* n = b2->f3; // n points to gv
9 }
```

Figure 4: pointer casting example in C

The second production on $\overline{\mathbf{F}}$ describes the inverse relationship of **F** (*i.e.,* $\mathbf{F}(a,b) \equiv \overline{\mathbf{F}}(b,a)$).

Productions 3, 4, 5, and 6 are **I-Alias**, which represent intermediate memory aliases that are encountered during the analysis. For instance, assume a and b are aliases (*i.e.,* they might point to the same object), which implies $\textbf{I-Alias}(a,b)$, then for $m \xleftarrow{Load} a$ and $b \xrightarrow{Load} n$ we can derive $\textbf{I-Alias}(m,n)$ through production 4, *i.e.,* m and n are also aliases.

Note that **I-Alias** is reflexive, *i.e.,* from $\textbf{I-Alias}(a,b)$, we can also derive $\textbf{I-Alias}(b,a)$ when we traverse the graph from b. In contrast, productions on **F** and $\overline{\mathbf{F}}$ are not. For example, deriving $\mathbf{F}(a_{gv}, n_1)$ represents a directional relationship where a_{gv} can flow to n_1 (and not the opposite direction).

To summarize, productions in L_R try to identify and match symmetric edges, *e.g., Store* **to** *Load*, $\overline{Load}$ **to** *Load*, $\overline{Field_{t,i,o}}$ **to** $Field_{t,i,o}$ (Be aware that there is no $\overline{Store}$ **to** *Store* or *Store* **to** $\overline{Store}$ since $a \xrightarrow{Store} n \xleftarrow{Store} b$ doesn't imply $\textbf{I-Alias}(a,b)$). More examples are prepared in Appendix A for reference.

4.2.2 Pointer Casting Handling

Pointer casting, as known as type casting, is a feature ignored by previous demand-driven alias analysis of C [59, 65], and yet it is critical to handle it properly to ensure soundness in a field-sensitive analysis.

All pointer casts are represented as *Assign* edges in PAG since pointer casting only changes the type instead the value of the source pointer. For convenience, we use $Cast_{T_1,T_2}$ to represent an *Assign* edge that casts a variable with T_1 type to a variable with T_2 type.

Consider an example in Figure 4 and its corresponding simplified PAG G_1:

$$a_{gv} \xrightarrow{Store} f_1 \xleftarrow{Field_{T_1,1,4}} b \xrightarrow{Assign} b_1 \xrightarrow{Cast_{T_1,T_2}} b_2 \xrightarrow{Field_{T_2,1,4}} f_3 \xrightarrow{Load} n$$

G_1: pointer casting

In Figure 4, it's easy to infer that the local pointer `t` points to `gv` since `b->f1` and `b2->f3` share the same memory object and field offset through `struct T1*` and `struct T2*` respectively.

In G_1, such pointer casting happens at the $Cast_{T_1,T_2}$ edge. Correspondingly, there are two different *Field* edges: $\overline{Field_{T_1,1,4}}$ and $Field_{T_2,1,4}$, before and after the cast. Instead of requiring the pair of *Field* edges to have the same type and field index, we model pointer casting by checking if they are

accessing the same relative offset (*e.g.,* o in $Field_{t,i,o}$), regardless of their base types. For example, in Figure 4, `f1` and `f3` both have a 4-byte offset, relative to `b` and `b2`, respectively.

Such normalization is sound because (1) all pointer casts are explicitly captured on PAG as *Cast* edges, (2) demand-driven rules in L_R already ensure variable-instance-level tracking so that the two base pointers are naturally pointing to the same object, *e.g.,* b_1 and b_2 in G_1, regardless of their types. It's worth mentioning that this strategy also naturally handles the `union` type and the `void*` type, as byte offset is consistent in an object no matter what type of pointers point to it.

4.2.3 Pointer Arithmetic Handling

Pointer arithmetic is performed by either pointer addition (+), pointer subtraction (-), array index (`[]`) (similar to addition and subtraction), or field access (->). All of them are represented as $Field_{t,i,o}$ edges in PAG. So far, L_R has not considered **pointer subtraction** and **non-constant pointer arithmetic**, which do occur frequently in Linux kernel. We describe our sound modeling below.

Pointer subtraction. So far, our productions on **I-Alias** assumed all *Field* edges are supposed to define field node fd_1 from a base node $base_1$. This is obviously the case for struct field accesses, as shown below:

$$\ldots \longleftarrow fd_1 \xleftarrow{Field_{T,1,4}} base_1 \xrightarrow{Field_{T,1,4}} fd_2 \longrightarrow \ldots$$

G_2: regular *Field* pair

We can see that from either direction we traverse the PAG, $Field_{T,1,4}$ pair in G_2 will match and **I-Alias**(fd_1, fd_2) or **I-Alias**(fd_2, fd_1) will be derived through production 6.

However, we note that there are pointer subtractions in the Linux kernel also, most commonly seen in `container_of()` [5], which was encountered 8,684 times during our analysis of Linux kernel v5.14. It is a construct that is not well modeled yet in the state-of-art alias analysis [36, 37]. Effectively, it inverts the "define" relationship such that a base node is defined through a field node. See the example below:

$$\ldots \longleftarrow fd_3 \xleftarrow{Field_{T,1,4}} base_2 \xleftarrow{Field_{T,-1,-4}} fd_4 \longleftarrow \ldots$$

G_3: pointer subtraction *Field* pair

In particular, fd_4 now defines $base_2$ through a *Field* edge. However, it is worth noting that it is accessed through a negative index (indicating a pointer subtraction). If we apply the original **I-Alias** in either direction, it is obvious that neither **I-Alias**(fd_3, fd_4) nor **I-Alias**(fd_4, fd_3) will be derived. Intuitively though, fd_3 and fd_4 should be considered aliases.

By observing the definition direction and pointer arithmetic nature, we introduce two new productions to L_R:

$$\textbf{I-Alias} \rightarrow Field_{t_1,i_1,-o}\ \textbf{I-Alias}\ Field_{t_2,i_2,o} \quad (7)$$

$$\textbf{I-Alias} \rightarrow \overline{Field_{t_1,i_1,o}}\ \textbf{I-Alias}\ \overline{Field_{t_2,i_2,-o}}\ \textbf{I-Alias} \quad (8)$$

These rules basically allow **I-Alias** to match two *Field* edges flowing in the same direction on the graph, provided that they share the same offset. Note that the first rule looks slightly different from the second, as the second has an additional non-terminal **I-Alias** at the end. It is necessary because (1) the overall graph traversal is by design directional, following the "define" directions, the last $\overline{Field}$ edge is a reverse edge that will effectively block the traversal; (2) the extra non-terminal **I-Alias** is essential for further tracking the aliases of the node to its left. For instance in G_3, besides **I-Alias**(fd_3, fd_4), we shall further derive more aliases of fd_3 through incoming "define" edges of fd_4, *e.g.,* **I-Alias**(fd_3, fd_5) through an extra edge of $fd_4 \xleftarrow{Assign} fd_5$.

Non-constant pointer arithmetic. Another exception is non-constant pointer arithmetic, as follows:

$$\ldots \longleftarrow fd_6 \xleftarrow{Field_{T,1,4}} base_3 \xrightarrow{Field_{T,v,o_v}} fd_7 \longrightarrow \ldots$$

G_4: non-constant *Field* pair

As shown in G_4, there's no concrete field index for the $Field_{T,v,o_v}$ edge, as it's potentially matching any previous or subsequent *Field* edges such as the $Field_{T,1,4}$. Thus, for soundness, we conservatively fall back to field-insensitive for such non-constant *Field* pairs, which intuitively means fd_7 may point to any fields of T.

More specifically, we always derive field pair productions 6, 7, 8 as long as either i_1 or i_2 is a non-constant index (or both are non-constants). In the interest of space, we put these productions (in total 4) to Appendix B and collectively refer to them as the non-constant **I-Alias** productions.

It is worth mentioning that such a conservative partial field-insensitive strategy will only slightly over-approximate field aliases on the derivation of the current **I-Alias**, representing a single pair of *Field* edges on the graph. There are many other constant field edge pairs on the graph, for which we still preserve the field sensitivity if matched. In practice, this means that we most likely still achieve reasonable precision from end to end.

In summary, we present additional productions in this section to formally and systematically handle pointer subtraction and non-constant pointer arithmetic, supporting widely used C features in the Linux kernel that are not previously supported in CFL-reachability-based alias analysis [59, 65].

4.3 Graph Traversal By Type Shortcuts

Even though the enhanced demand-drive analysis (DDA) is precise, we find it is not scalable for most global variables (see §6.1). In this section, we describe a novel solution to integrate the spirit of type-based analysis into DDA (which are by design complementary), by introducing two **type-based shortcuts** on the graph to address the scalability challenge, while still preserving soundness. It is worth noting that the solution is designed to allow for flexible adjustment to balance scalability and precision.

Recalling type-based alias analysis (TBA), which has the advantage of being more scalable by directly over-approximating the aliases based on types. In the context of

PAG graph traversal, TBA avoids the expensive edge-by-edge traversal. However, the downside of TBA is its imprecision. In particular, we find that the traditional TBA is extremely imprecise for most global variables because they are of primitive types, have been converted to primitive types, or their primitive type fields have been address-taken (see §6.1).

However, we realize that TBA can in fact be generalized and integrated elegantly into DDA. We show that such a hybrid approach achieves much better scalability than DDA-only and much better precision than TBA-only. This indicates that the two approaches can be highly complementary when combined in the right way.

Insights. Our solution is based on two insights: First, when integrating TBA into DDA, we have the flexibility to switch between TBA and DDA back and forth. This means that we can combine the benefits of both approaches, retaining some precision of DDA and improving scalability by resorting to TBA on demand. Second, because of the flexibility, we no longer need to find pointer aliases directly based on the type of the global variable itself (which is what traditional TBA did [17, 30]. In such cases, if a_{gv} is a primitive pointer type, say `int*`, all `int*` pointers will be aliases of a_{gv}.) Instead, we can anchor the TBA on a much less common type (*e.g.,* a composite type) by seeking an opportunity during DDA, *i.e.,* when propagates to the field of a composite type (via a *Store*).

4.3.1 Type-based Field-To-Field Shortcut

To illustrate the above insights more clearly, consider the following PAG G_5:

$$a_{gv} \xrightarrow{Store} f_1 \xleftarrow{Field_{T,1,4}} base_1 \rightarrow ... \rightarrow base_2 \xrightarrow{Field_{T,1,4}} f_2 \xrightarrow{Load} n$$

G_5: PAG of a long propagation chain

In G_5, a_{gv} may flow to many nodes through $base_1$ and $base_2$. Such propagation may go across many function calls and even syscalls (due to the flow-insensitive nature of the graph). Thus, it can be costly to derive **I-Alias**$(base_1, base_2)$. On the other hand, traditional TBA is not precise as a_{gv} may be a common pointer type, say `int*`, such that all `int*` pointers will be aliases with a_{gv}.

As a result, instead of edge-by-edge graph traversal with DDA, we assume **I-Alias**(f_1, f_2) can be directly derived as they share the same base type and field index. For now, we assume there is no pointer casting and the type is t according to the $Field_{t,i,o}$ edge; later we relax this assumption in §4.3.2. More generally, for *Field* related productions 6, 7, 8 and their non-constant versions (in Appendix B), we consider the **I-Alias** between *Field* edge pairs can be directly derived; again, we assume $t_1 = t_2$ and $i_1 = i_2$ on those edge pairs in productions 6, 7, 8 for now.

PAG transformation. To implement the above strategy, we will need to transform a PAG by creating explicit shortcut edges according to corresponding productions to enable traversing such edges.

For example, upon seeing a $fd_s \xleftarrow{Field_{t,i,o}} base_s$ edge, which matches the first $\overline{Field}$ edge in production 6 and 8, we will search globally on the graph for the second *Field* edge according to these two productions and their non-constant versions. Specifically, we would

(1) create a type shortcut $fd_s \xrightarrow{Shortcut_f} fd_x$ for a matching $base_x \xrightarrow{Field_{t,i,o}} fd_x$ (required in production 6) or $base_x \xrightarrow{Field_{t,v,ov}} fd_x$ (required in the non-constant version of production 6).

(2) create a type shortcut $fd_s \xrightarrow{Shortcut_b} fd_x$ for a matching $base_x \xleftarrow{Field_{t,i,-o}} fd_x$ (required in production 8) or $base_x \xleftarrow{Field_{t,v,ov}} fd_x$ (required in the non-constant version of production 8).

(3) disconnect all involved $\overline{Field}$ and *Field* edges. This final step is to ensure that the choice of DDA and TBA is mutually exclusive (*i.e.,* redundant to follow both). The transformation of G_5 will result in G_6 as follows:

$$a_{gv} \xrightarrow{Store} fd_1 \xrightarrow{Shortcut_f} fd_2 \xrightarrow{Load} n$$

G_6: Creating a type shortcut edge in G_5

Now we simply connect fd_1 to fd_2 with a "field-to-field shortcut" edge. This way, we can successfully derive $\mathbf{F}(a_{gv}, n)$, by taking the initial *Store* (DDA), the $Shortcut_f$ (TBA), and then the *Load* edge (DDA).

Rule changes to incorporate TBA into DDA. Together with the graph transformation, we introduce two additional productions to L_R:

$$\textbf{I-Alias} \rightarrow Shortcut_f \quad (9)$$

$$\textbf{I-Alias} \rightarrow Shortcut_b \ \textbf{I-Alias} \quad (10)$$

These productions simply allow a field-to-field shortcut edge to be treated as an edge that connects two intermediate aliases (fields) as in production 6 and 8 respectively (and their non-constant versions). This is basically all we need to cleanly integrate the TBA fully into DDA.

Soundness consideration. We describe two considerations for soundness. First, for PAG transformation, when we meet a *Field* edge, we either create shortcut edges to all eligible target nodes or none. This is because by creating and following a selected few shortcut edges, we might lose aliases. Second, in addition to applying the shortcut rules, we also will apply other non-shortcut rules (*i.e.,* DDA) for the same fd node. For example, in G_6, if there is an additional edge $fd1 \xrightarrow{Assign} fd3$, it must also be traversed to ensure that no potential aliases are missed. This is naturally followed by the definition of CFL-reachability where all encountered edges will be evaluated against all productions.

Tunable design. Second, we do have the flexibility to choose whether to perform PAG transformation whenever we meet a $\overline{Field_{t,i,o}}$ edge. In some circumstances, even though our analysis is anchored on a composite type, it may still be a popular type used widely (*e.g.,* `struct list_node`), leading

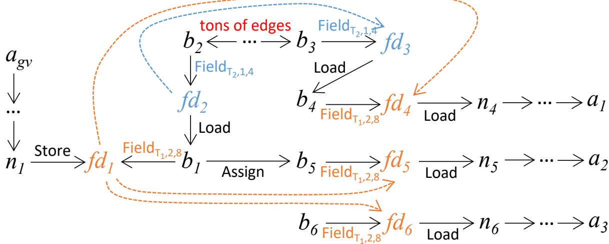

Figure 5: An example PAG illustrating how TBA into DDA are integrated, dashes are potential shortcut edges

to too many corresponding shortcut edges if we decide to transform the PAG. For such cases, we can choose DDA instead. In other words, our design is tunable and can cover a wide range of DDA+TBA strategies. In the extreme case, if we never decide to create any shortcuts, the analysis falls back to DDA-only. Note that there does not exist any fallback to traditional TBA-only analysis, since shortcuts are based on field edges as opposed to arbitrary pointer nodes.

An example. We use a slightly more complex example in Figure 5 to better illustrate the above aspects. Assuming there are no shortcut edges initially. If we apply TBA directly, all nodes that share the same type with a_{gv} will be imprecisely considered as pointer aliases. If we apply DDA directly from a_{gv}, it will run into scalability issues when exploring the tons of edges between b_2 and b_3. When attempting to create shortcuts by transforming the PAG, we are presented with the first opportunity when DDA encounters $fd_1 \xleftarrow{Field_{T_1,2,8}} b_1$ edge. If we decide to transform PAG for this edge, it will create three shortcut edges, as there are three corresponding $Field_{T_1,2,8}$ edges. According to the productions, all three edges should be traversed as they all can potentially be aliases.

Alternatively, we can choose to forgo the first opportunity and continue DDA until it meets the second opportunity: $fd_2 \xleftarrow{Field_{T_2,1,4}} b_2$. In that case, we see only one shortcut edge will be created, which indicates that T_2 is a less common type. However, if decide to switch to TBA and create a shortcut edge, it means we can skip only the edges between $\overline{Field_{T_2,1,4}}$ and $Field_{T_2,1,4}$ (they are disconnected after PAG transformation); there exists another branch from b_1 to b_5 (*i.e.,* an *Assign* edge), which we still need to explore using DDA. This ensures that we do not miss any potential alias.

Note that taking different opportunities will result in different precision and scalability tradeoffs, as different decisions will result in different pointer alias set *pas* results shown in Figure 5. We will evaluate such tradeoffs in §6.1.

Generally speaking, taking shortcuts may reduce precision. However, our analysis must follow DDA before and after taking shortcuts, which naturally alleviates such precision reduction. For example, even if we made an over-approximation by taking the shortcut between fd_1 and fd_4, it still requires that the edges between a_{gv}-fd_1 and those between fd_4-a_1 to match the DDA rules. Otherwise, a_1 will not be considered a pointer alias of gv, as the end-to-end path traversal fails.

4.3.2 Precise Field-To-Castsite Shortcut

In the last section, we introduced how type-based field-to-field shortcut works without considering pointer casting. In this section, we relax this assumption with a novel strategy by creating and following "Field-To-Castsite" shortcut edges. We will show how the solution is *much more precise* than state-of-art type-based solutions and still preserves soundness.

Type-based solutions like TYPM [36] and MLTA [37] address pointer casting by collecting a "cast map" for every pointer type that has been cast to/from another type. For example in MLTA, during its pre-processing stage, it collects all pointer castings, and conservatively considers all pointers of `T2*` as aliases with pointers of `T1*`. Such a cast map will significantly over-approximate the real aliases as it applies globally to all pointers. The problem is exacerbated when the type "escapes" to a primitive type, *e.g.,* `void*`. This is because the cast map relationship is transitive. If `T1*` has been cast from/to `void*`, and `void*` has been cast from/to `T2*` and `T3*`, then pointers with `T1*`, `T2*`, `T3*`, and `void*` will all be aliased.

To incorporate TBA into DDA, one obvious method is to create shortcut edges when we find a pair of *Field* edges where (1) their types are "compatible" according to the cast map [36, 37], and (2) the two *Field* edges have matching normalized offsets. Unfortunately, this significantly over-approximates aliases as explained before.

Next, we describe our novel solution that is much more precise and sound at the same time.

Insights. Different from cast-map-based approaches, which simply consider all "compatible" types of pointers (*i.e.,* nodes on PAG) as aliases, we propose to consider only the pointers that have actually encountered cast instructions to be aliases. For instance, if we observed only a single cast edge between type `T1*` and `T2*`, *e.g.,* $n_1 \xrightarrow{Cast_{T_1,T_2}} n_2$ on PAG, we will not consider all nodes with types `T1*` or `T2*` as aliases like in cast-map-based approaches. Instead, we only consider n_1 and n_2 as aliases. Fundamentally, we are able to make such a differentiation because of the data-flow capability provided by DDA. In other words, we have more information (from PAG) that can inform a much more precise TBA decision.

Figure 6 illustrates the insight more clearly. Using the state-of-art cast-map-based approach [36, 37], all nodes, from f_1 to f_5, are considered aliases. In our newly proposed approach, only f_1, f_2, and f_3 are considered aliases because only they are attached to base nodes that have experienced type casts between `T1*` and `T2*`. Ideally, we should connect these field nodes directly. However, as we see in the figure, a matching $Field_{T_2,1,4}$ edge may not follow immediately after the type cast (see n_3 and f_3). Therefore, for soundness, instead of creating a "field-to-field" edge as we did before, we choose to

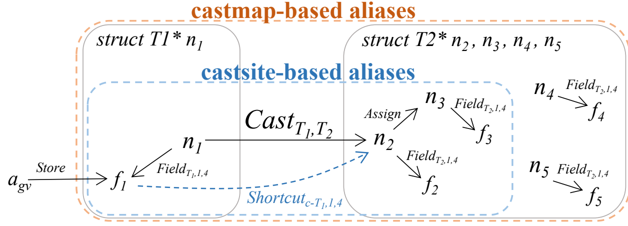

Figure 6: An example PAG illustrating how castsite-based aliases are more precise than castmap-based

create a special "field-to-castsite" shortcut edge, which retains the type, index, and offset information of the original $\overline{Field_{T_1,1,4}}$ edge, denoted as $Shortcut_{c-T_1,1,4}$.

PAG transformation. Similar to §4.3.1, we aim to transform a PAG such that the original *Field* related DDA productions 6, 7, and 8 and their non-constant versions can go through shortcuts instead. Intuitively, the shortcut edge replaces the first *Field* edge in an original DDA production and keeps searching for the second matching *Field* edge. More generally, when encountering a $\overline{Field_{t,i,o}}$ edge or a $Field_{t,i,-o}$ edge, we shall create a $Shortcut_{c-T_1,i,o}$ edge (note that we normalize the direction of the edge and the positive/negative sign of its offset). The edge is between "the field node of the first *Field* edge" (f_1 in Figure 6) to "the nodes of type T_2 attached to $Cast_{T_1,T_2}$ or $Cast_{T_2,T_1}$ edges" (n_2). This way, we effectively allow the $Shortcut_c$ to reach castsites and use DDA to explore all potential aliases of the original node. Note that if we decide to perform the PAG transformation, we will also disconnect the *Field* edge (*e.g.,* $Field_{T_1,1,4}$ in the example) to ensure the choice of DDA and TBA is mutually exclusive.

Rule changes to incorporate TBA into DDA. To accommodate the above changes, we introduce two additional productions to L_R:

$$\textbf{I-Alias} \rightarrow Shortcut_{c-t_1,i_1,o}\ \textbf{I-Alias}\ Field_{t_2,i_2,o} \quad (11)$$

$$\textbf{I-Alias} \rightarrow Shortcut_{c-t_1,i_1,o}\ \textbf{I-Alias}\ \overline{Field_{t_2,i_2,o}}\ \textbf{I-Alias} \quad (12)$$

These two productions are effectively pointer-casting-aware versions of productions 9 and 10. In both productions 11 and 12, we can see the **I-Alias** non-terminal after $Shortcut_{c-t_1,i_1,o}$ representing the query of aliases of the destination node (*e.g.,* n_2 in Figure 6) at castsite. In production 12, we see an extra **I-Alias** non-terminal at the end similar to production 10. This is for the same reason to account for pointer subtraction cases.

Note that we omit the non-constant versions of the productions for brevity. Basically, we would allow the above productions to directly derive if the offset in the shortcut edge or the one in the *Field* edge is non-constant.

4.3.3 Soundness

As our enhanced demand-driven graph traversal follows the principle of Andersen-style alias analysis, by conservatively modeling advanced pointer manipulation features in C, we will show that the shortcut design over-approximates DDA and is therefore also sound.

Assumptions. (1) We assume all global variables are always initially used through their names in the source code; for example, there should not be intended or unintended out-of-bound writes to the global variables. (2) We assume all pointer operations are captured as edges (*e.g., Field* edges) on PAG.

Proof. Under the DDA grammar (without shortcuts), given a PAG G and arbitrary nodes a, b such that $\mathbf{F}(a,b)$ holds through path p.

For any derivation of production 6 (which we use as an example) on p, w.l.o.g. the PAG will take the form as below:

$$a \rightarrow \ldots \rightarrow fd_i \xleftarrow{Field_{t_1,i_1,o}} b_i \xleftrightarrow{\textbf{I-Alias}} b_k \xrightarrow{Field_{t_2,i_2,o}} fd_k \rightarrow \ldots \rightarrow b$$

First, according to the definition of shortcuts, with a matching pair of $\overline{Field_{t_1,i_1,o}}$ and $Field_{t_2,i_2,o}$ edges, we will always connect fd_i to fd_k directly, if t_1 is the same as t_2. In such a case, we simply skip over the intermediate **I-Alias** edges on the graph and still derive $\textbf{I-Alias}(fd_i, fd_k)$ as DDA would.

If t_1 is different from t_2, it is not hard to see that there must exist a $Cast_{t_1,t_2}$ edge (or multiple cast edges with intermediate types) between b_i and b_k. According to the production 11, we will jump to the castsite (*i.e.,* destination node of the $Cast_{t_1,t_2}$ edge). Further, the production will continue to search for an **I-Alias** before ultimately finding the matching $Field_{t_2,i_2,o}$ edge, and successfully derive $\textbf{I-Alias}(fd_i, fd_k)$ as well.

Second, since we only disconnect involved *Field* edges after PAG transformation, if there are other edges connected to fd_i or fd_k, they will still be explored, just as in DDA.

Together, we show that the shortcut strategy will always over-approximate and never create side effects that prevent certain edges from being traversed. Note that the example we gave assumes production 6 is the one that matches the PAG. However, we note that the logic applies similarly to productions 7 and 8, as well as the non-constant versions.

5 Implementation

We implement Unias based on LLVM-13.0.1 and SVF-2.1, with 2,500 lines of C++ code. We also implement auxiliary components which are specific to the application of global variable protection, with 400 additional lines of C++ code.

Algorithm. To implement Unias, we give a worklist algorithm after PAG transformation in Algorithm 1. This algorithm encompasses productions 1 to 6 and also includes corresponding shortcuts for production 6. Due to the space constraints, we omit the pointer arithmetic productions and shortcuts in the algorithm as they share similar insights as production 6. In our actual implementation, we do handle all productions from 1 and 12 and the non-constant versions of them.

In the algorithm, `Propagate` is used to maintain the worklist, and the recursive function `ComputeAlias` can be used for both **F** query and **I-Alias** query. The *state* represents the state of the alias analysis. There are two possible states, S_f

Algorithm 1: Algorithm of Unias

```
1  Propagate(worklist, cur, src, state):
2      if <cur, state> ∉ Reach(src)
3          Reach(src) ← Reach(src) ∪ {<n, state>}
4          worklist ← worklist ∪ <cur, src, state>
5  ComputeAlias(node, state):
6      Alias(node) ← ∅
7      w ← {<node, node, state>}
8      while w ≠ ∅ do
9          remove <c, s, state> from w
10         Alias(s) ← Alias(s) ∪ {c}
11         Alias(c) ← Alias(c) ∪ {s}
12         if state == S_b
13             for each c <--Assign-- n
14                 Propagate(w, n, s, S_b)
15             for each c <--Load-- n
16                 ComputeAlias(n, S_b)
17                 for each a ∈ Alias(n)
18                     for each a <--Store-- m
19                         Propagate(w, m, s, S_b)
20                     for each a --Load--> m
21                         Propagate(w, m, s, S_f)
22             for each c <--Field_{t,i,o1}-- p
23                 ComputeAlias(p, S_b)
24                 for each a ∈ Alias(p)
25                     for each a --Field_{t,i,o2}--> fd
26                         if o1 == o2
27                             Propagate(w, fd, s, S_f)
28             for each c --Shortcut_f--> fd
29                 Propagate(w, fd, s, S_f)
30             for each c --Shortcut_b--> fd
31                 ComputeAlias (fd, S_b)
32                 for each a ∈ Alias(fd)
33                     Propagate(w, fd, s, S_f)
34             for each c --Shortcut_{c-t,i,o1}--> castnode
35                 ComputeAlias (castnode, S_b)
36                 for each a ∈ Alias(castnode)
37                     for each a --Field_{t,i,o2}--> fd
38                         if o1 == o2
39                             Propagate(w, a, s, S_f)
40         for each c --Assign--> n
41             Propagate(w, n, s, S_f)
42         for each c --Store--> n
43             ComputeAlias(n, S_b)
44             for each a ∈ Alias(n)
45                 for each a --Load--> m
46                     Propagate(w, m, s, S_f)
```

and S_b, which represent the algorithm is to match forward edges (*e.g.,* **F**) and backward edges (*e.g.,* $\overline{\mathbf{F}}$ and the beginning of **I-Alias**) respectively.

Thus, given a global variable address a_{gv}, querying its flows-to is equal to invoke `ComputeAlias` (a_{gv}, S_f), and for those intermediate aliases introduced during the query, we recursively invoke `ComputeAlias` for them with state S_b, as line 16, 23, 31, 35 and 43 show, and change the state to S_f when about to match forward edges, *e.g., Load* at line 20.

Call graph generation. We apply the state-of-the-art solution MLTA [37] due to its sound and precise nature.

Init function recovery. We find that only a subset of kernel initialization functions are labeled properly (*e.g.,* `__init`, `__exit`, or `module_init`). This means that there are missing labels for many other functions that are invoked only once during kernel initialization. This will cause us to under-estimate the `ro_after_init` global variables. To identify the "missing" init functions, we conduct a conservative fixed-point analysis on the call graph generated by MLTA. The analysis will label new init functions based on the existing init function set. Specifically, if a function is called by only init functions, it will be put in the init function set. This results in 629 more init functions compared to the 2,064 labeled in the Linux kernel v5.14 with `defconfig` [1].

Annotating global variables. Once we identify read-only and `ro_after_init` global variables, we implement an LLVM pass to set the attributes of such global variables directly to protect them, *i.e.,* `gv->setSection(".data..ro_after_init");`

6 Evaluation

To demonstrate Unias's contribution to safeguarding global variables, we analyze all global variables in a specific Linux kernel to identify those qualified as `ro_after_init`, then we verify the results and show their accuracy. In addition, we conduct a case study to show how most recent kernel exploits that targeted global variables can be mitigated by either `ro_after_init` or SFI. More specifically, we aim to answer the following questions: (1) How does Unias achieve a better tradeoff between precision and scalability compared to pure DDA and pure TBA for the purpose of protecting global variables? (§6.1) (2) Is the result truly sound? (§6.2) (3) Does the result help with real-world attacks? (§6.3)

Dataset. For alias analysis, we qualify global variables in our scope by following rules, (1) the global variable is in a write-able section (*e.g.,* not exist in read-only section) (2) the global variable is not used in assembly code files (*e.g.,* files with suffix .S or .s) – 92 global variables are excluded here. Finally, we have 12,089 global variables in our analysis scope.

There are 69,285 fields in these global variables, where 65,960 fields belong to 8,764 composite global variables, and 3,325 fields belong to non-composite (pointer and primitive types) global variables. Note that arrays are considered as one

Table 1: Overall `ro_after_init` Results Comparison

	Unias		DDA		Steensgaard		TBA	
#finish-gv	12,089	100.0%	2,911	24.1%	12,089	100.0%	12,089	100.0%
#ro_a_i[1]	5,091	42.1%	1,639	13.6%	1,243	10.3%	917	7.6%
#finish-fd	69,285	100.0%	12,934	18.7%	69,285	100.0%	69,285	100.0%
#field	60,903	87.9%	10,483	15.1%	4,639	6.7%	4,290	6.2%
#funcptr	13,796	97.0%	5,642	39.7%	2,492	17.5%	3,875	27.2%
#data-fd	47,107	85.6%	4,841	8.8%	2,147	3.9%	415	0.8%
total time[2]	560		> 9,716		46		< 1	

[1] #ro_a_i: # of global variables determined to be `ro_after_init`
[2] The unit of measurement for the total time is the CPU hour

element. Among the 69,285 fields, there are 14,224 control data (*e.g.,* function pointers) fields and 55,061 non-control data (*e.g.,* integers, data pointers) fields.

Experiment setup. All experiments are conducted on a machine with an AMD EPYC 7542 32-Core CPU and 2TB of RAM, running Ubuntu 20.04.5 LTS. The version of LLVM is 13.0.1. And the target Linux kernel version is v5.14. The kernel LLVM IR is compiled by Clang 13.0.1 with `defconfig` and optimization level "-O0".

To demonstrate the effectiveness of Unias, we compare it against the state-of-art sound alias analysis (§3.2), namely Andersen-style demand-driven alias analysis (DDA), whole-program type-based alias analysis (TBA), and whole-program Steensgaard's alias analysis (Steensgaard). To be fair, all analyses are field-sensitive, flow-insensitive, and context-insensitive. Besides, we use the same time limit for Unias and DDA since they both start with a demand-driven strategy. Specifically, each global variable is analyzed separately with a time limit of one hour. If the analysis does not complete in time, we conservatively view the variable as not eligible for protection. As for TBA and Steensgaard, there is no time limit since both of them are scalable whole-program analyses that will finish much quicker than Unias and DDA.

Note that Unias is tunable, as mentioned in §4.3.1, and we must choose a configuration to run it. Specifically, whenever we encounter a $\overline{Field_{t,i,o}}$ edge, we have the choice of either creating and taking the shortcut edges or continuing DDA. We choose the following simple configuration for our main results: if the number of shortcut edges we need to create in meeting a a $\overline{Field_{t,i,o}}$ edge is less than 200 (threshold), we will choose to create and take these shortcuts. Otherwise, we continue with DDA. One can obtain the number of $Shortcut_f$ and $Shorcut_c$ edges for a given $\overline{Field_{t,i,o}}$ edge before the decision. Generally speaking, a higher threshold means taking shortcuts more aggressively and will result in more scalability but less precision. Later on, we also evaluate other thresholds.

6.1 Results

Table 1 shows the results with respect to `ro_after_init`. We can see Unias clearly performs much better than the other methods. First, it achieves a comprehensive analysis of 100% of global variables. Second, it finds 5,091 (#`ro_a_i`) global variables protectable as `ro_after_init` at the object granularity (42.1% of all global variables) and 60,903 (#field) at the field granularity (87.9%). If we look at the function pointer (#func-ptr) and data pointer fields (#data-fd), which are common targets of exploits, Unias finds 97.0% and 85.6% of them protectable. Note that the current `ro_after_init` mechanism operates only at the object level. However, the field-level results can still be used in other fine-grained integrity protection mechanisms (which we discuss in §6.3).

Table 2: Scalability Results of pure DDA

	< 30m	< 1h	< 2h	< 4h	< 8h
#finished-gv	15.4%	24.1%	34.2%	41.1%	50.2%
#ro_a_i	9.0%	13.6%	19.4%	24.0%	27.7%
#finished-fd	11.5%	18.7%	29.7%	39.9%	48.6%
#field	9.1%	15.1%	25.7%	33.4%	41.9%
#funcptr	20.7%	39.7%	55.9%	61.4%	65.0%
#data-fd	6.1%	8.8%	18.1%	26.1%	36.0%

Unias vs pure DDA. As shown in Table 1, Unias can finish analyzing all of the 12,089 global variables whereas DDA can finish only 2,911 (24.1%), both under the same one-hour time limit for each global variable. Since those unfinished global variables are considered unprotectable, Unias identifies significantly more protectable global variables — about 2 times more object-wise and 5 times more field-wise.

Interestingly, we find 144 global variables that are considered protectable by pure DDA but considered unprotectable by Unias with the same 1-hour budget. This is expected because DDA is the most precise, provided that it can finish the analysis within the time limit. Unfortunately, we cannot predict whether DDA can finish within an hour beforehand. It is an interesting future work to tune Unias automatically by taking into account the time budget and history of performance. One might think it is possible to perform a two-phase analysis, *i.e.,* pure DDA first to see which global variables are difficult to finish and then run Unias on those. However, under such a strategy, the time taken by the initial run of DDA-only should still count towards the overall analysis time and therefore this overall may not always be the better strategy.

To comprehensively evaluate the result of pure DDA, we also measure the results with other time budgets, including 30 mins, 2 hours, 4 hours, and 8 hours, for each global variable. The results are shown in Table 2. While we do see improvements as the time budget increases, we see still about half of the global variables unfinished even after 8 hours. This clearly represents a significant scalability challenge to pure DDA.

Unias vs Steensgaard. In Table 1, we see Steensgaard finishes analyzing all global variables in 46 hours, which is fairly quick and only slower than TBA. However, we find that only 10% global variables are considered protectable due to the significant over-approximation by design. We can see that it is still much less precise compared to Unias and Andersen-style DDA. It only protects 2.7% more global variables and 0.5% more fields compared to TBA.

Table 3: Tunablility Result of Unias

threshold*	<1000	<200	<50
#finished	12,089	12,089	11,381
#ro_a_i	4,672	5,091	4,912
#field	54,911	60,903	59,034
min time	<1s	<1s	<1s
max time	16m32s	45m57s	59m20s
avg time	1m15s	2m47s	4m39s

* threshold < 1000 means creating and taking shortcut when the corresponding shortcut edges number is less than 1000

Unias vs pure TBA. As expected, TBA has excellent scalability, finishing the analysis of all global variables in only 10 minutes. However, when it comes to precision, TBA's results are extremely poor, as only 917 (7.6%) of global variables and 4,290 (6.2%) fields are considered protectable. We find all protectable ones share the same attributes: (1) are composite types, (2) never have their fields address-taken, (3) never cast from/to non-composite types. However, the majority of global variables do not share these attributes (*e.g.,* a large fraction of global variables are primitive types). Because of the severe limitation of TBA, we find no new protectable global variables compared with Unias.

Tunability. To demonstrate the tunable design of Unias, we run Unias with different thresholds following the simple strategy described earlier. As shown in Table 3, a higher threshold means that we will choose to take more shortcuts, which will result in generally better scalability but lower precision since we spend less time on DDA. On the other hand, a lower threshold means that we will choose to take fewer shortcuts and result in generally worse scalability and higher precision. Notably, choosing a threshold of "50" leads to 708 unfinished global variables in 1 hour, and subsequently, fewer protectable global variables compared to results with a threshold of "200".

6.2 Accuracy

In this section, we compare Unias against two types of ground truth to understand the accuracy of Unias. We may miss the opportunity to protect a global variable if we have false positives (FP) in alias analysis, *i.e.,* false pointer aliases that lead to store operations. Conversely, we may falsely protect a global variable if we have false negatives (FN) in alias analysis, *i.e.,* miss pointer aliases that lead to store operations.

Missing protections (FP of alias analysis). As mentioned in §3.2, kernel developers have manually labeled 127 global variables as the `ro_after_init` (`defconfig` in v5.14 Linux kernel). We treat them as ground truth and perform a separate run on these 127 global variables to check whether Unias missed any. Note that the ones we miss will not lead to error; instead, we simply lose a few protectable global variables.

Overall, 14 global variables were found unprotectable. It means we successfully identified the rest 113 (89%) protectable global variables, showing good coverage. Interestingly, 7 of them are due to the over-approximation of pointer aliases. The remaining 7 are due to missing labels of init functions, which prevent us from excluding the write operations that occurred in such functions.

False protections (FN of alias analysis). Since we do not have pre-existing ground truth about global variables in writable sections of the Linux kernel, we rely on dynamic traces of load/store instructions generated from fuzzing. Specifically, we run the compiled kernel with all 5,091 global variables labeled as `ro_after_init` in QEMU. We then instrument all load/store instructions to trace their operations on the 12,089 global variables. Specifically, we log both the global variable and its field (according to the target address of the load/store instructions). In addition, we log the binary address of the load/store instruction itself (EIP).

To fuzz the kernel, we use the state-of-art Linux kernel fuzzer Syzkaller [21], with its built-in syscall descriptions as well as descriptions generated by SyzDescribe [25] to get as much coverage as we could. With 2 weeks of fuzzing (2 cores $\times$ 8 vms). We find 645 global variables are stored and 3,142 global variables are loaded at least once after kernel initialization, which shows a considerable coverage (out of 12,089 global variables in total).

Encouragingly, our result finds no false protection if we use the Unias results at the object granularity. In other words, the kernel never raised an error by mislabeled "read-only" global variables. However, when we inspect the results at the field granularity, we realize that there are actually 34 fields from 29 global variables that experienced write instructions. The reason that the 29 global variables did not cause any trouble during fuzzing is that there are writes to other fields of them which are also captured by Unias before.

Through our manual investigation of a sampled set of 13 global variables (out of the 29), we found all of the corresponding missing store-to-field cases are due to missing pointer information in LLVM or PAG. As we mentioned in §4.3.3, Unias is sound by design and is based on the assumption that the PAG captures all the pointer operations. Specifically, we divide the reasons into two major categories: assembly code and pointer-to-integer.

(1) Assembly code. We found 9 global variables' fields are mistakenly considered protectable because their addresses leaked into the assembly code, causing our analysis to miss their propagation. Unfortunately, assembly code does not get compiled into LLVM IR and therefore is missed in PAG as well. One potential strategy to handle assembly code is to treat any global variables whose addresses propagate to assembly code as unprotectable. This strategy is conservative and sound. However, it may increase false positives substantially. One other strategy is to model the caller of assembly code or lift them systematically into LLVM. We plan to explore such options in future work.

(2) Pointer-to-integer. We found 4 global variables' fields are mistakenly considered protectable because of this reason.

Table 4: Case Study

CVE	global variable	protection
CVE-2017-5123	mmap_min_addr	SFI
	dac_mmap_min_addr	SFI
CVE-2019-2215	selinux_enforcing	SFI
CVE-2022-22057		SFI
CVE-2021-27365	iscsi_iser_transport	ro_after_init
CVE-2022-1786	misc_format	ro_after_init
CVE-2021-42008	modprobe_path	SFI
CVE-2021-43267		SFI
CVE-2022-0185		SFI
CVE-2022-27666		SFI
CVE-2022-32250		SFI
CVE-2022-34918		SFI
CVE-2023-0179		SFI
CVE-2023-32233		SFI

This is due to the Linux kernel sometimes casting pointers to `long` integers and further performing pointer arithmetic on those integers [15] and later cast them back to pointer types (evident by the use of `ptrtoint`/`inttoptr` in LLVM IR). Even though this is permitted in C, it is considered risky – a quote from the C standards: "the result is implementation-defined" [8]. By design, PAG captures only pointer-related operations and not integer arithmetic. To handle these cases, we need to extend the PAG so that we can keep track of such integer arithmetic. At a minimum, we would need to connect the source node before it is cast to an integer and the destination node after it is cast back to a pointer type, without keeping track of the integer arithmetic in between. In essence, we can create a special *Field* edge connecting the source and destination node (which may or may not be the same type). This will allow DDA to track the flows-to relationship at least with some precision. We envision a simple analysis that can ensure soundness: (1) allowing the source and destination node to be matched even if their types differ, and (2) conservatively considering all fields that the destination pointer points to as aliases with the source pointer. We leave a detailed implementation and analysis to future work.

6.3 Case Study

To show how results of Unias can be beneficial, we collect 37 publicly available Linux kernel exploits [12, 54, 57] from 2017 to 2023. As shown in Table 4, we find 13 exploits against 13 CVEs (35% of all CVEs) targeting 6 unique global variables. This suggests a significant focus on exploiting kernel vulnerabilities through global variables. Out of these 6 global variables, Unias finds that `misc_format` and `iscsi_iser_transport`[1] should not be modified (*i.e.,* no intended store instruction) after kernel initialization. We manually verify the results to be correct, and thus it can be safely labeled as `ro_after_init`. On the other hand, Unias did find intended store instructions for the remaining 4 global variables and our manual confirmation verified their correctness. This implies that these variables are not eligible for protection `ro_after_init`. Nevertheless, these 4 global variables are still protectable via SFI. Specifically, we discovered that none of the store instructions used in the exploit were part of the intended set identified by Unias, implying that the exploit can be successfully mitigated by denying such write instructions at runtime via SFI. Furthermore, Unias accurately identified all intended store instructions, resulting in zero missed cases. As a result, we can confidently assert that we will not encounter any errors caused by mistakenly denying legitimate access. Take `modprobe_path` as an example, we find 19 intended store instructions in total, all stemming from the `SYSCTL` feature. Besides SFI, developers can choose to disable the feature so that `modprobe_path` becomes `ro_after_init`. Indeed, we find that on some Samsung Android devices, this feature is disabled (likely for security reasons).

[1] Note that we need to add `CONFIG_INFINIBAND_ISER=y` to the kernel config in order for the variable to be included.

7 Discussion

Further precision improvements. There are two main sources of precision losses: (1) flow-insensitive and context-insensitive in DDA, and (2) type-based shortcuts through TBA. For (1), while flow-insensitivity is a necessity to ensure soundness, context-insensitivity is not and can be potentially incorporated in Unias, *e.g.,* by duplicating copies of the same function at each call site on PAG. However, the tradeoff is decreased scalability. For (2), we can investigate other strategies to decide when to take shortcuts. For example, instead of using a fixed threshold at every decision point, we can vary it depending on more precise estimates, *e.g.,* by looking ahead and summarizing future edges in some way. We can even evaluate multiple "choices" in parallel to make more informed decisions.

Further scalability improvements. First, since there are many intermediate aliases that are repeatedly computed for different global variables, we can cache such intermediate results across queries. Second, we envision our solution can be applied incrementally for Linux kernel releases (similar to [61]). This means even more caching can be done across different kernel versions.

General alias analysis for other applications. Even though Unias is applied to finding pointer aliases of global variables, the nature of our solution generalizes beyond it. For example, we can find pointer aliases of stack and heap objects that are allocated in one module but may propagate to another module. This can be applied to program modularization and isolation [11, 28, 32, 36, 40–43]. We hinted in §6.3 that we can use our technique to precisely restrict write accesses to sensitive data [9, 19, 47, 48] (*e.g.,* `uid` field (since our analysis is field-sensitive) in `struct cred` [50]).

Another potential application is to apply our analysis to resolve indirect calls in the Linux kernel – instead of tracking how the addresses of global variables propagate, we would track how the addresses of functions propagate. We believe our analysis can potentially outperform or complement the state-of-the-art [37]. We leave these interesting applications for future exploration.

8 Related Work

Points-to and alias analysis. As described throughout the paper, there exists a large body of alias analysis work, which can be broadly classified into data-flow-based [10, 55, 59, 65] and type-based [17, 37]. We discussed their advantages and disadvantages in the paper and how they can be combined extensively. We now discuss a few key related works on demand-driven alias analyses of C, and show how they differ from our demand-drive version. Many demand-driven alias analyses of C have achieved limited precision and soundness guarantees due to the modeling of a subset of features of the C language [26, 65]. Specifically, the analysis in [65] is field-insensitive without considering pointer types and pointer arithmetic hence is much less precise than ours. Wang *et al.* [55] proposed a solution to transform the Linux kernel program into graphs that are amendable to existing graph systems. Such systems offer scalable processing of large graphs. We regard their solution as orthogonal to ours as they do not fundamentally change the complexity of an alias analysis, but graph systems' performant implementations can potentially further improve the speed of our hybrid alias analysis. The analysis in [26] is field-sensitive but may not sound due to ignoring pointer casting and pointer arithmetic. Later, a more precise demand-driven pointer analysis [59] is proposed with field-, flow-, context- and even partial path-sensitivity. However, the analysis cannot be soundly scaled to analyze the Linux kernel, because achieving both flow-sensitivity and soundness given a multi-entry program is hard (see §3.2).

Static analysis of the Linux kernel. A large body of work has applied static analysis on the Linux kernel for various purposes, such as bug finding and security enforcement [20, 34, 37–39, 56, 60–62]. The majority include some form of data-flow-based alias analysis and do not provide soundness guarantees. This is because they do not track soundly all the pointer aliases of a given object, irrespective of whether the object is a stack-allocated, heap-allocated, or globally-defined object. The key difficulty is that the addresses of these objects can propagate to global objects across syscalls. The difficulty is exactly what we address in this paper. SUTURE [62] is the only work that attempts to reason about aliases across syscalls. However, its flow-sensitive analysis will have to consider all possible sequences of syscall invocations to be sound. In addition, it does not handle the case where the address of one global object propagates to another global object, leading to unsound results. Lastly, we have seen some recent work on improving the precision of type-based alias analysis by considering multiple layers of types [37, 63]. However, both focused on function pointers instead of data pointers.

Protection and attack of global variables. A few prior works have proposed defenses to protect the integrity of global variables in the Linux kernel. Xiao *et al.* [58] studies a variety of global variables that control important security mechanisms in the Linux kernel, such as auditing framework, AppArmor, and NULL pointer dereference mitigation. They subsequently classify these variables into different types for protection, *e.g.,* read-only. However, the work simply assumed the labels given in the kernel without inferring any additional global variables, which would require a proper alias analysis. Song *et al.* [47] proposes a principled defense to enforce data flow integrity on critical data in the Linux kernel, by mediating write accesses to such data (including 1,490 global variables). However, in order to identify all the write instructions, a sound alias analysis is required. Otherwise, legitimate writes through pointer aliases may be mistakenly blocked.

9 Conclusion

This paper presents a novel and general hybrid alias analysis based on formalized algorithms. Our solution elegantly unifies the data-flow-based and type-based strategies elegantly, which are complementary by nature. We show that it is effective against an extremely challenging target, *i.e.,* the Linux kernel, using the application of global variable protection.

References

[1] https://www.linux.org/threads/the-linux-kernel-configuring-the-kernel-part-1.8745/.

[2] Introduce post-init read-only memory. https://lwn.net/Articles/676145/.

[3] LLVM. https://llvm.org/docs/index.html.

[4] SVFIR or Program Assignment Graph (PAG). https://github.com/svf-tools/SVF/wiki/Technical-documentation#12-svfir-or-program-assignment-graph-pag.

[5] Type safe(r) list_entry replacement: container_of. https://lwn.net/Articles/5482/.

[6] New reliable android kernel root exploitation techniques. http://powerofcommunity.net/poc2016/x82.pdf, 2016.

[7] Defeating samsung knox with zero privilege. https://infocondb.org/con/black-hat/black-hat-usa-2017/defeating-samsung-knox-with-zero-privilege, 2017.

[8] ISO/IEC 9899:2018, Information technology — Programming languages — C. 2018.

[9] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008.

[10] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[11] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *USENIX annual technical conference*, volume 2010. Boston, 2010.

[12] Ofer Chen. CVE-2023-32233-PoC. https://github.com/oferchen/POC-CVE-2023-32233, 2023.

[13] Quan Chen, Ahmed M Azab, Guruprasad Ganesh, and Peng Ning. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 167–178, 2017.

[14] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1184, 2020.

[15] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[16] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *NDSS*, 2017.

[17] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, 1998.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[19] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *USENIX Security Symposium*, pages 1037–1054, 2021.

[20] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[21] Google. Syzkaller. https://github.com/google/syzkaller, 2022.

[22] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 26, 2017.

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[24] Hackthebox. CVE-2022-0185 Writeup. https://www.hackthebox.com/blog/CVE-2022-0185:_A_case_study, 2022.

[25] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3262–3278. IEEE Computer Society, 2023.

[26] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, 2001.

[27] Grant Hernandez. CVE-2019-2215 Writeup. https://hernan.de/blog/tailoring-cve-2019-2215-to-achieve-root/. 2022.

[28] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. {KSplit}: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, 2022.

[29] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[30] Iain Ireland, José Nelson Amaral, Raúl Silvera, and Shimin Cui. Safetype: Detecting type violations for type-basedalias analysis of c. *Softw. Pract. Exper.*, 2016.

[31] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Tagbleed: breaking kaslr on the isolated kernel address space using tagged tlbs. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 309–321. IEEE, 2020.

[32] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved

system dependability via virtual machines. In *OSDI*, volume 4, pages 17–30, 2004.

[33] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1963–1976, 2022.

[34] Changming Liu, Yaohui Chen, and Long Lu. Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel. In *NDSS*, 2021.

[35] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting missed security operations through differential checking of object-based similar paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1627–1644, 2021.

[36] Kangjie Lu. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1610–1624. IEEE Computer Society, 2023.

[37] Kangjie Lu and Hong Hu. 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*, 2019.

[38] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932, 2016.

[39] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr. checker: A soundy analysis for linux kernel drivers. In *USENIX Security Symposium*, pages 1007–1024, 2017.

[40] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.

[41] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakc. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[42] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. Lxds: Towards isolation of kernel subsystems. In *USENIX Annual Technical Conference*, pages 269–284, 2019.

[43] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 157–171, 2020.

[44] Samsung Knox News. Real-time Kernel Protection (RKP). https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp, 2016.

[45] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xmp: Selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577. IEEE, 2020.

[46] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

[47] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[48] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[49] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *ACM SIGPLAN Notices*, 40(10):59–76, 2005.

[50] Abhinav Srivastava and Jonathon Giffin. Efficient protection of kernel data structures via object partitioning. In *Proceedings of the 28th annual computer security applications conference*, pages 429–438, 2012.

[51] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[52] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[53] Theori. CVE-2022-32250 Writeup. https://blog.theori.io/research/CVE-2022-32250-linux-kernel-lpe-2022/, 2022.

[54] TurtleARM. CVE-2023-0179-PoC. https://github.com/TurtleARM/CVE-2023-0179-PoC, 2023.

[55] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News*, 45(1):389–404, 2017.

[56] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with {KINT}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 163–177, 2012.

[57] Xairy. Linux kernel exploitation. https://github.com/xairy/linux-kernel-exploitation.

[58] Jidong Xiao, Hai Huang, and Haining Wang. Kernel data attack is a realistic security threat. In *International Conference on Security and Privacy in Communication Systems*, pages 135–154. Springer, 2015.

[59] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 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*, pages 327–337, 2018.

[60] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel. ESEC/FSE 2020, 2020.

[61] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V Krishnamurthy, Trent Jaeger, et al. Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel. In *2022 Network and Distributed System Security Symposium*, 2022.

[62] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2021.

[63] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

```
1  int o_gv;
2
3  struct T{
4      int* fd0;
5      int* fd1;
6  }o_base;
7
8  store_load(){
9      int* n3 = &o_gv;
10     int** n4 = &n3;
11     int** n5 = n4;
12     int* n6 = *n5;
13 }
14
15 field(struct T* base1){
16     base1->fd1 = &o_gv;
17     struct T* base2 = base1;
18     int* n7 = base2->fd1;
19 }
20
21 field(&o_base);
```

Figure 7: example for production rules

[64] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Surya Nepal, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. {SoftTRR}: Protect page tables against rowhammer attacks using software-only target row refresh. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 399–414, 2022.

[65] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, 2008.

[66] Xiaochen Zou. CVE-2022-27666 Writeup. https://etenal.me/archives/1825, 2022.

A More Examples on Applying Basic Production Rules

To better understand how the production rules introduced in §4.2.1 work, we give examples for different edges in Figure 7.

Recalling §4.2.1, any global variable (*e.g.*, `o_gv`) will only be used through its address, denoted as a_{gv}. To simplify the following PAGs, we have **S** as *Store*, **L** as *Load*, **A** as *Assign*, and $\mathbf{Fd}_{t,i,o}$ as $Field_{t,i,o}$.

For *Store* and *Load* edges, consider codes in function `store_load()`, which PAG will be:

$$a_{gv} \xrightarrow{\mathbf{A}_1} n_3 \xrightarrow{\mathbf{S}} n_4 \xrightarrow{\mathbf{A}_2} n_5 \xrightarrow{\mathbf{L}} n_6$$

G_7: PAG of `field()`

In function `store_load()`, `n3` and `n6` are both pointer aliases of `o_gv`, as $\mathbf{F}(a_{gv}, n_3)$ and $\mathbf{F}(a_{gv}, n_6)$ should both hold on G_7. The derivation will be:

a_{gv} $(\mathbf{F})$
$\rightarrow$ a_{gv} $(\mathbf{A}_1\ n_3\ \mathbf{F})$
$\rightarrow$ a_{gv} $\mathbf{A}_1\ n_3$ $(\mathbf{S}\ n_4\ \textbf{I-Alias}_1\ \mathbf{L})$
$\rightarrow$ a_{gv} $\mathbf{A}_1\ n_3\ \mathbf{S}\ n_4$ $(\overline{\mathbf{F}}\ \textbf{I-Alias}_2\ \mathbf{F})\ \mathbf{L}$

$\rightarrow\ a_{gv}\ \mathbf{A}_1\ n_3\ \mathbf{S}\ n_4\ (\varepsilon\ \varepsilon\ \mathbf{F})\ \mathbf{L}$
$\rightarrow\ a_{gv}\ \mathbf{A}_1\ n_3\ \mathbf{S}\ n_4\ (\mathbf{A}_2)\ n_5\ \mathbf{L}$
$\rightarrow\ a_{gv}\ \mathbf{A}_1\ n_3\ \mathbf{S}\ n_4\ \mathbf{A}_2\ n_5\ \mathbf{L}\ n_6$

During the derivation, as $\textbf{I-Alias}_1(n_4, n_5)$ holds, it means n_5 is an alias of n_4, which is also easily observed as `int** n5 = n4` in `store_load()`. This truth shows the intuition of production 4, *i.e.,* if a_{gv} or its alias (*e.g.,* n_3) is **stored** to *somewhere* (*e.g.,* n_4), we should find all *somewhere*'s aliases (*e.g.,* n_5), then **load** from them and get all a_{gv}'s aliases (*e.g.,* n_6). This procedure could perform recursively like multiple layers of intermediate alias $\textbf{I-Alias}_n$ as there might be multiple times of **store** and correspondingly **load**, and for each $\textbf{I-Alias}_n$ layer caused by **store**, we should find all the aliases of this layer, then continue analysis through their **load** edges.

Finally we have derived that `n3` and `n6` are pointer aliases of `o_gv`, as $\mathbf{F}(a_{gv}, n_3)$ and $\mathbf{F}(a_{gv}, n_6)$ hold.

$\textbf{I-Alias}_1(n_4,n_5)$ shown in G_7 describes one kind of derivations ($\varepsilon\ \varepsilon\ \mathbf{F}$) of **I-Alias**. To better understand other productions of **I-Alias**, we give function `field()` in Figure 7 and its PAG G_8, which shows the derivation of field edges:

$$a_{gv} \xrightarrow{\mathbf{S}} fd1_1 \xleftarrow{\mathbf{Fd}_{T,1,4}} base_1 \xrightarrow{\mathbf{A}} base_2 \xrightarrow{\mathbf{Fd}_{T,1,4}} fd1_2 \xrightarrow{\mathbf{L}} n_7$$

G_8: PAG of `field()`

In `field()`, it's intuitive to figure out the global variable address `&o_gv` will finally flow to local pointer `n7` in function `func`, which means `n7` will point to `o_gv`, as $\mathbf{F}(a_{gv}, n_7)$ should hold on G_8.

First we could easily derive $\textbf{I-Alias}(base_1,base_2)$ in G_8 just as $\textbf{I-Alias}_1(n_4,n_5)$ in G_7. Then we could derive $\textbf{I-Alias}(fd1_1,fd1_2)$ through production 6 since $fd1_1$ and $fd1_2$ are defined through the same $Field_{T,1,4}$ edges from $base_1$ and $base_2$ respectively. As $base_1$ and $base_2$ are aliases for each other, $fd1_1$ and $fd1_2$ should also be aliases for each other. Finally we can derive $\mathbf{F}(a_{gv},n_7)$, as we have $\textbf{I-Alias}(fd1_1,fd1_2)$ with production 4.

Notice that $fd1_1$ is not an alias of $base_1$, while they both point to the same object o_{base} but with different field indexes. We decouple such field/base alias relationship of an object. Such separation reflects the nature of field-sensitive, *i.e.,* differentiate every field of struct objects.

B Non-Constant Pointer Arithmetic Rules

$\textbf{I-Alias} \rightarrow\ (Field_{t_1,v,o_v} \mid \overline{Field_{t_1,v,o_v}})\ \textbf{I-Alias}\ Field_{t_2,i_2,o}$
$\textbf{I-Alias} \rightarrow\ (Field_{t_1,v,o_v} \mid \overline{Field_{t_1,v,o_v}})\ \textbf{I-Alias}\ \overline{Field_{t_2,i_2,o}}\ \textbf{I-Alias}$
$\textbf{I-Alias} \rightarrow\ (Field_{t_1,i_1,o} \mid \overline{Field_{t_1,i_1,o}})\ \textbf{I-Alias}\ Field_{t_2,v,o_v}$
$\textbf{I-Alias} \rightarrow\ (Field_{t_1,i_1,o} \mid \overline{Field_{t_1,i_1,o}})\ \textbf{I-Alias}\ \overline{Field_{t_2,v,o_v}}\ \textbf{I-Alias}$

Generally speaking, all of these non-constant pointer arithmetic productions conservatively match the pair of *Field* edges once there's a non-constant index (v) in either or both of them. Still, we add extra non-terminal **I-Alias** for the same reason in §4.2.3 (*i.e.,* uncertain definition directions).