Seal: Towards Diverse Specification Inference for Linux Interfaces from Security Patches

Wei Chen
The Hong Kong University of Science and Technology, China
wchenbt@cse.ust.hk

Bowen Zhang*
The Hong Kong University of Science and Technology, China
bzhangbr@cse.ust.hk

Chengpeng Wang
Purdue University, USA
wang6590@purdue.edu

Wensheng Tang
The Hong Kong University of Science and Technology, China
wtangae@cse.ust.hk

Charles Zhang
The Hong Kong University of Science and Technology, China
charlesz@cse.ust.hk

Abstract

Linux utilizes interfaces as communication protocols across different subsystems while ensuring manageability. These interfaces standardize interactions between various subsystems; however, the absence of complete calling contexts can result in the mishandling of data from other entities, i.e., interaction data, thus incurring vulnerabilities. Even worse, the effectiveness of static bug detectors could be severely hindered due to the lack of interface specifications. Previous solutions, seeking to automate the inference of interface specifications, are tailored to a subset of the interaction data behavior and, hence are deficient in generalizability.

This research presents Seal, a framework that leverages security patches to achieve the automatic inference of diverse interface specifications. Those specifications, formulated as value-flow properties, could adeptly characterize interaction data behaviors for individual interfaces and the synergistic relationships among multiple interfaces. Technically, Seal assesses the impact of code changes in program dependencies, abstracts specifications from changed value-flow paths, and detects bugs via reachability analysis. Experiments show Seal attains a precision of 71.9% and the specifications could accommodate various bug types. We utilized Seal to identify 167 unseen bugs in Linux, hidden for an average of 7.7 years. So far, 95 of them are confirmed by Linux maintainers, 56 of which fixed by our patches.

*Corresponding author.

CCS Concepts: • **Security and privacy** → **Operating systems security**.

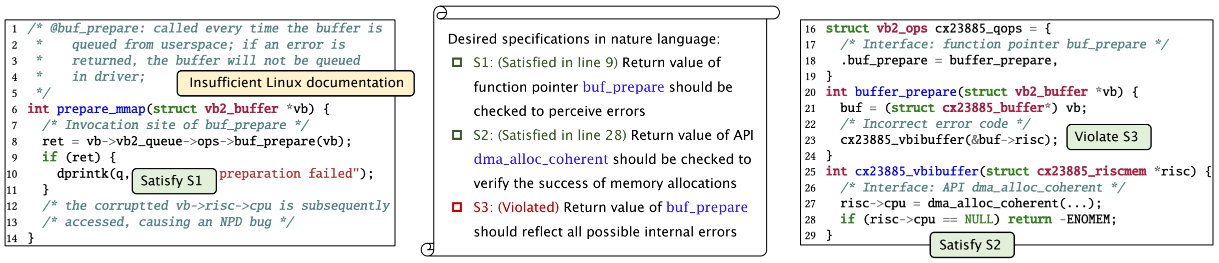
Keywords: Linux, Specification Inference, Patch Analysis, Static Bug Detection

ACM Reference Format:
Wei Chen, Bowen Zhang, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2025. Seal: Towards Diverse Specification Inference for Linux Interfaces from Security Patches. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717487

1 Introduction

The monolithic kernel of Linux integrates numerous subsystems, e.g., drivers, network protocols, and file systems, that are developed by a global community to deliver various services to the user space. To ensure developers from different parties can efficiently collaborate on such a vast codebase, the kernel uses interfaces to specify the functions available for interactions. First, one form of interfaces, i.e., APIs, encapsulates frequently utilized functionalities. Second, Linux incorporates another form of interfaces, i.e., function pointers, to manage common operations among subsystems. For instance, in Fig. 1, `prepare_map` invokes the function pointer `buf_prepare` at line 8 without the need to understand its implementation `buffer_prepare` at line 20 where the API `dma_alloc_coherent` is used for allocating DMA memory regions at line 27.

Such programming idioms standardize and expedite the parallel development of the Linux core and various subsystems. However, it's difficult for developers when using or implementing interfaces to fully obey latent data interaction rules. For instance, developers could forget to verify the nullness of incoming parameters or to convey erroneous error codes. Since in the real-world Linux development, only type safety is strictly assured, these latent data interaction protocols often result in system crashes and exploitable vulnerabilities. In Fig. 1, the type signature of `buffer_prepare` is guaranteed to conform to the declaration of `buf_prepare`,



```
1  /* @buf_prepare: called every time the buffer is
2   *    queued from userspace; if an error is
3   *    returned, the buffer will not be queued
4   *    in driver;
5   */
6  int prepare_mmap(struct vb2_buffer *vb) {
7    /* Invocation site of buf_prepare */
8    ret = vb->vb2_queue->ops->buf_prepare(vb);
9    if (ret) {
10     dprintk(q,            preparation failed");
11   }
12   /* the corruptted vb->risc->cpu is subsequently
13   /* accessed, causing an NPD bug */
14 }
```

Desired specifications in nature language:

- S1: (Satisfied in line 9) Return value of function pointer buf_prepare should be checked to perceive errors
- S2: (Satisfied in line 28) Return value of API dma_alloc_coherent should be checked to verify the success of memory allocations
- S3: (Violated) Return value of buf_prepare should reflect all possible internal errors

```
16 struct vb2_ops cx23885_qops = {
17   /* Interface: function pointer buf_prepare */
18   .buf_prepare = buffer_prepare,
19 }
20 int buffer_prepare(struct vb2_buffer *vb) {
21   buf = (struct cx23885_buffer*) vb;
22   /* Incorrect error code */
23   cx23885_vbibuffer(&buf->risc);
24 }
25 int cx23885_vbibuffer(struct cx23885_riscmem *risc) {
26   /* Interface: API dma_alloc_coherent */
27   risc->cpu = dma_alloc_coherent(...);
28   if (risc->cpu == NULL) return -ENOMEM;
29 }
```

Figure 1. Examples of two forms of interfaces: the API `dma_alloc_coherent` (line 27), and the function pointer `buf_prepare` (line 18). Incorrect error code in line 18 violates the desired specification S3, causing an NPD bug.

whereas the behaviors of interaction data, `vb` and the return value of `buffer_prepare`, cannot be automatically understood by computers. Subsequently, the erroneous error code at line 23 causes the caller, `prepare_mmap`, to inadvertently access the corrupted memory region and trigger a Null Pointer Dereference (NPD) bug.

Ideally, we expect that the anticipated behaviors of interaction data are thoroughly documented as specifications [79, 97] to guide developers and static analyzers, examplified by rules S1 to S3 in Fig. 1. Unfortunately, in practice, interface specifications are often documented poorly. In lines 1-5 of Fig. 1, the Linux documentation only explains the functionality of `buf_prepare` [14], omitting how to process internal errors, e.g., API failure in line 28 [9]. On the other hand, manually summarizing interface specifications is extremely challenging even for kernel experts since the specifications require multiple parties to determine, not to mention the excessive number of interfaces and diverse behaviors of the interaction data. An example is whether a user input should be checked uniformly or per implementation of interface `fb_check_var` has been constantly controversial [1].

We can opt to automatically infer specifications [47, 48, 51, 53, 82, 91]. However, these approaches fail to generate comprehensive and precise specifications for a large number of interfaces. Specifically, one line of work [31, 33, 39, 53, 82] observes that certain code patterns in the codebase would reveal specifications. However, these patterns are often empirically determined, overfitting code samples under inspection and failing to adapt to other scenarios. Another line [16, 34, 40, 46, 48, 51, 91, 95] cross-checks multiple functional-similar codes and picks out the majority as specifications. Likewise, the comparison metrics are tailored to certain behaviors and by nature probabilistic, degrading the effectiveness. Our experiments show that CRIX [51], which cross-checks the conditional statements in the peer slices of critical variables to identify missing-check bugs, only detects one of the bugs found by us. Existing attempts [47] have also employed security patches to infer API post-handling specifications, but the form to represent specifications, 4-tuples, suffers from limited expressiveness.

This paper determined to unleash the potential of security patches to infer specifications with much more diversity. Security patches [77] serve as strong proof of specification violations and the changes from buggy code to patched one shed light on the correct way to manipulate interaction data. However, the inference process is non-trivial due to the challenges from three aspects. First, the behaviors of interaction data to be constrained are irregular, and in a more complicated case, multiple interfaces may interact synergistically, thus the specifications must be adequately expressive to capture these varied behaviors. Second, security patches can introduce flexible and occasionally subtle syntactic changes when fixing different bugs, making impact analysis and root cause understanding difficult. Third, the scalability problem would arise once employing highly precise static analysis on the large Linux codebase.

Our Approach. We introduce Seal to incrementally infer diverse specifications for Linux interfaces with security patches. Our insight is that, by formalizing specifications as value-flow properties [61], we could harness their descriptive power to characterize complex relationships among interaction data. Years of static analysis research [61, 62, 68, 69] has demonstrated that value-flow properties are capable of inspecting a very broad category of vulnerabilities. The inferred properties that focus on values and uses of critical variables are sufficiently abstract. We utilize them to identify violations inside other implementations and usages of the same interface, which are expected to adhere to the same latent rules of manipulating interaction data. For instance, the patch in Fig. 3 fixes the NPD in Fig. 1 by conveying the return value in line 23, from which we could learn the reachability relation between the error code and the return value of interface is vital. Other implementations of the function pointer `buf_prepare` that invoke `dma_alloc_coherent` should obey the specification as well.

Technically, Seal leverages field-, flow-, context-, and path-sensitive analysis to build program dependence graphs as the central data structure for specification inference and violation detection. Our prototype gathers the changed value-flow paths inter-procedurally based on the data-, control-, and flow-dependence changes of interaction data, and utilizes them as resources to abstract value-flow properties. Notably, our framework could also be easily extended to incorporate more program properties to achieve a more profound analysis of code changes. Afterward, the specifications are utilized for bug detection by searching for realizable value-flow paths in other code regions.

We evaluated our prototype on Linux v6.2 with 12,571 security patches and obtained inspiring results. Seal successfully uncovers 167 Linux bugs, 95 bugs were confirmed by maintainers, and 56 of them were fixed by our patches. Remarkably, our found bugs are long-latent, hiding for an average of 7.7 years, and can inflict various security impacts. Besides, our bug detection achieves a precision of 71.9%, outperforming existing patch-based and deviation-based approaches [47, 51]. We believe that other systems that extensively use interfaces could also benefit from Seal. In summary, the main contributions of this paper are:

- We propose an expressive formulation for interface specifications, enabling the description of complex behaviors of interaction data.
- We implement an extensive framework, Seal, that takes security patches as inputs to infer interface specifications.
- We conduct a systematic evaluation to demonstrate that our approach can achieve high versatility and pinpoint numbers of previously unknown bugs in Linux.

2 Background

This section explains interface specifications and the consequences of violations (§ 2.1). We emphasize the challenges in obtaining interface specifications thereafter (§ 2.2).

2.1 Interface Specifications

Linux integrates two forms of interfaces, i.e., function pointers and APIs, to effectively manage multiple subsystems without increasing development and maintenance expenses. Initially, the developers of Linux core designed and prepared interfaces, which were then increasingly implemented and utilized by subsystem developers. During collaborations among various entities, extensive data are exchanged between invokers and implementers of interfaces. In this paper, we term the variables and memory regions that are defined and initialized by one party but also visible and used by another as ***interaction data***.

Crucially, interaction data comes with inherent requirements that must be adhered to during manipulation. However, when only local contextual information is available, these rules may become obscure, leading to inadvertent mishandling of interaction data and resulting in vulnerabilities. Interface specifications emerge to clarify the proper usage of interaction data when invoking or implementing interfaces.

Specifications for Invoking APIs. Linux encapsulates commonly used functionalities into APIs to avoid code duplication. For instance, platform driver developers could invoke API `platform_device_register` [13] to register current devices. The interaction data associated with APIs include 1) parameters to be passed, 2) return values that typically indicate the execution results, and 3) side effects that necessitate subsequent operations [38, 47, 91]. Interface specifications explain the pre- and post-conditions when invoking APIs, detailing the acceptable values for parameters, the meanings of return values [41], the coherence among multiple APIs, and necessary post operations to perform. Violating these rules would inflict high severe vulnerabilities. For example, without noticing APIs freeing the passing parameters, subsequent access to the parameter would lead to use-after-free [86]. Failing to use paired deallocation APIs for cleaning up allocated variables can result in memory leaks [33, 38, 56, 70].

Specifications for Implementing Function Pointers. Linux abstracts common operations among a set of subsystems into function pointers using these instead of specific implementations to enhance compatibility and extensibility. Subsystem developers are responsible for implementing function pointers to handle unique features. Still for platform drivers, the structure `struct platform_driver` specifies a set of function pointers to be implemented [13], such as `probe` for platform device insertion. The interaction data for function pointers subsume 1) incoming arguments, 2) return values, and 3) accessible global variables [57]. When implementing function pointers, the specifications explain which checks should be performed on incoming arguments and global variables, as well as expected return values to synchronize the internal status. Likewise, violating these specifications can lead to a range of consequences. First, arguments and global variables could come from user space or hardware and thus can carry malicious data [55, 94]. Without sufficient checks, missing-check bugs [19, 51, 55] could occur. Second, prior efforts have discussed that incorrect error codes would hide internal errors from the kernel and cause bugs like memory corruptions [31, 39, 73].

2.2 Lack of Specifications

In practice, interface specifications could be defined as type signatures, annotations [8], and documentation [10–12] to guide programmers. For instance, in the Linux kernel, specifications can be enforced through assertions using macros such as `BUG_ON` or `WARN_ON` [6]. These specifications are designed to regulate interaction data from various aspects, including acceptable types, runtime values validated at compilation time, and complicated data usages that require sophisticated static analyzers [2, 19, 31, 62, 78, 85].

Unfortunately, specifications are often lacking. In practice, Linux documentation often focuses on explaining interface functionalities but tends to ignore the rules of interaction data manipulation. Furthermore, manually enumerating adequate specifications is extremely challenging even for kernel experts, due to the vast interface spaces and irregular interaction data behavior.

Large Interface Space. First, the massive codebase of Linux involves numerous subsystems, each developing its own interfaces for specialized purposes. As a result, manual summarization requires extensive domain knowledge and significant developers to be involved. Specifically, kernel APIs provide functionalities ranging from memory management [33, 53], reference counting [38], to locking [23, 45], etc. For function pointers, Linux v6.2 contains 82,549 indirect calls, taking charge of functionalities such as initialization, handling interrupts and system calls, etc. The existing effort, Archerfish [90], highlights the challenges in documenting numerous interrupt handlers in Linux and resorts to large language models (LLMs) to model their semantics.

Irregular Behaviors. Second, interaction data behaviors are diverse and irregular. As we delineated in § 2.1, the behaviors encompass valid values, conditions for uses, and appropriate actions for error handling, among others. Later, we will discuss how previous works often summarize specifications for only a subset of behaviors, thereby providing limited generalizability. More importantly, multiple interfaces would synergistically communicate with each other. A representative example is function pointer implementations often invoke APIs to achieve functionality. Consequently, the specifications should be determined collaboratively.

Remark. In summary, comprehensively summarizing interface specifications is extremely labor-intensive. Worse yet, interface behaviors could be "unstable", and specifications could be proactively changing through commits and discussions. This makes manual settlement of these pieces of knowledge fall through. Thus, an automatic approach to liberate human labor is urgently demanded.

3 Overview

This section begins by discussing cutting-edge techniques and their shortcomings (§ 3.1). On that basis, we summarize challenges to infer specifications from patches (§ 3.2) and state the insight of our patch-based solution (§ 3.3).

3.1 Limitations of Existing Efforts

To address the inadequacies of interface specifications and reduce human interventions, recent years have witnessed three lines of solutions. They typically leverage two kinds of inputs, i.e., codebases, and security patches, aiming to automatically summarize interface specifications, however, in limited scopes. As a result, none of these methods could generate comprehensive and precise specifications for various interfaces. We discuss the designs and shortcomings of these categories, followed by our choices below.

Pattern-based Approach. The first category [31, 33, 39–41, 53, 82] observes that certain code patterns would reveal the semantics and correct usages of interfaces. They manually summarize code patterns as templates and match them inside the codebase for specifications. For instance, Goshawk [53] noticed that custom memory management functions must eventually invoke primitive functions. Apex [40] studied the characteristics of error paths to collect error specifications. K-MELD [33] exploited the pairwise relationships between specialized allocation/deallocation APIs. While these efforts achieve high precision, the code patterns cannot be migrated since the code patterns are empirically determined and tend to overfit manually reviewed code samples.

Deviation-based Approach. The second category [16, 34, 48, 51, 57, 91] holds the common belief that for one interface, the majority of usages are correct, while deviations are potentially buggy. These methods often propose a clustering metric to group comparable codes and employs a similarity algorithm to identify their commonalities. For instance, APISan [91] represents API usages as symbolic traces and statistically counts the majority by occurrence. Juxta [57] symbolically compares multiple implementations of VFS entry functions in file systems to derive latent high-level semantic rules. Although these approaches significantly advance the scope of supported interfaces, they suffer from low precision and recall. First, the underlying assumption that the majority of usages are correct does not always hold, let alone the inaccuracies introduced by the probabilistic clustering and similarity algorithms. Second, these clustering and similarity algorithms are tailored to certain behaviors thus cannot be easily adapted to other contexts.

Patch-based Approach. Patch-based solutions infer specifications from the additions and deletions of code in security patches [72]. The code differences illuminate how bugs are manifested and more crucially, reveal the correct usages. For instance, APHP [47] models API post-handling specifications as a four-tuple, i.e., *<target API, post-operation, critical variable, path condition>*, and extracts these components from code differences and patch descriptions. The inferred specifications are reliable as patches provide strong evidence of specification violations. However, this proposed specification form, by definition, is limited to describing only one type of interaction data behavior.

Our Choice. In summary, existing techniques have not sufficiently addressed the versatility required to describe complex interaction data behaviors. This work aims to bridge this gap by comprehensively characterizing the correct behaviors of interaction data while maintaining reasonable recall, precision, and scalability. We have chosen to use security patches as inputs for deriving specifications, recognizing their potential to elucidate proper interaction data usage across a broad spectrum of interfaces.

3.2 Problem Challenges

While security patches are valuable for mining interface specifications, the inference process is non-trivial due to the challenges from restricted patch information and the intractable specification design.

Empirical Study. To characterize patches that fix mishandling of interaction data, we conducted an empirical study by constructing a dataset of 158 historical Linux patches [3].

The dataset forms 32 groups, where bugs in each group originate from the same mishandling error when implementing or using the same interface. Inspired by change impact analysis [15], we adopt slicing techniques [65], complemented with manual inspection to locate bug traces for study.

C1. Restrictive Patch Information. First, we noticed that security patches could make subtle code changes, i.e., less than 10 lines in extreme cases, to fix violations. This localized perspective impedes a thorough and precise understanding of bug behavior, thereby complicating the inference of specifications. Our study indicates that only 34.8% of bug traces are confined within patched functions. For instance, the patch in Fig. 3 simply corrects the return value of `buffer_prepare`, yet identifying the root cause, the failure of `dma_alloc_coherent`, is crucial. Furthermore, the liberal use of pointers results in indirect data flows that are often overlooked in the absence of precise alias analysis [43]. Therefore, analysis with high sensitivity is required to locate statements affected by code changes to avoid false negatives. Meanwhile, the analysis should not involve too many irrelevant statements that hurt precision.

C2. Intractable Specification Design. Second, the specifications must be expressive yet sufficiently abstract to accommodate various bug types and code customizations. Our dataset contains 11 bug types, including memory corruption bugs, taint-style bugs, etc, and covering all interaction data mishandling errors detailed in § 2.1, underlining the need for versatile specifications. We have also evaluated APHP on our dataset and found only 7 groups of bugs (19.6%) could be reported due to the restrictive specification form for API post-handlings. Besides, although bugs in each group share the same mishandling error, their bug traces are dispersed across various functions, involve numerous intermediate statements, and are sparsely located. Consequently, the specifications must exclude program elements particular to input patches to be general enough.

3.3 Insight of Seal

Seal deduces interface specifications from security patches and conducts violation detection to identify bugs. Specifically, we advocate reflecting code changes to differences in value-flow paths and abstracting value-flow properties [61] from them to compose interface specifications. The design conquers the aforementioned challenges from two aspects.

First, value-flow paths describe how the value of one variable flows along statements. These paths could carry out rich information, e.g., path conditions, and control flow relationships. We can comprehensively perceive changes in the dependencies of interaction data through value-flow path changes, paving the way to extract interface specifications. For instance, path additions or removals often imply correct values of interaction data or necessary post-operations.

Second, value-flow properties feature our specifications high degree of versatility and abstraction. These properties

$$
\begin{array}{rl}
\text{Specification } S := & i\ [arg^i * \rightarrow ret^i *]\ \ q* \\
\text{Quantifier Constraint } Q := & (\forall | \exists | \nexists)_{v|u}\ :\ r^+ \\
\text{Path Relation } R := & v \overset{c}{\hookrightarrow} u \mid u_1 \prec\ u_2 \mid r_1 \land r_2 \mid r_1 \lor r_2 \mid \neg\, r \\
\text{Value } V := & arg^i \mid ret^f \mid g \mid l \mid v.\texttt{field} \\
\text{Use } U := & arg^f \mid ret^i \mid g \mid deref \mid div \mid \ldots \\
\text{Condition } C := & e_1 < e_2 \mid e_1 > e_2 \mid e_1 == e_2 \\
 & \mid c_1 \lor c_2 \mid c_1 \land c_2 \mid \neg\, c \\
\text{Expression } E := & e_1 + e_2 \mid e_1 - e_2 \mid\ * v \\
 & \mid \textbf{call}\ f(v, \ldots) \mid v \mid\ \ldots \\
\text{Argument } arg \in Arg & \text{Return } ret \in Ret \\
\text{Global } g \in G & \text{Literal } l \in L \\
\text{Function Pointer } i \in I & \text{API } f \in F
\end{array}
$$

Figure 2. Syntax of interface specification.

aggregate multiple value-flow paths and specify the predicates concerning their existence and coherence, e.g. *must exist*. When evaluating value-flow properties, only the sources, sinks, and predicates matter, while intermediate statements propagated along the value-flow paths are disregarded. Prior efforts in static analysis [25, 35, 69] have demonstrated value-flow properties could underpin the detection of a wide range of vulnerabilities, including memory safety bugs, resource leaks, and taint-style bugs.

4 Interface Specification

Before delving into Seal, this section gives a formal definition of interface specifications designed based on value-flow properties (§ 4.1), followed by three examples to demonstrate the expressiveness of proposed formulation (§ 4.2).

4.1 Specification Formulation

We formalize the interface specification in Fig. 2, which essentially describes the constraints over interaction data that each implementation of a function pointer and/or API usage needs to satisfy. The specifications intertwine two kinds of interfaces. In what follows, we systematically elaborate on our fundamental principles.

For an interface that declares a function pointer i with arguments arg^i and return values ret^i, its specification consists of several constraints. Each quantifier constraint $q \in Q$ specifies the path relationships R that should be obeyed among multiple critical "interaction data" and their "uses". A value $v \in V$ represents regulated incoming data for an interface, including function pointer arguments arg^i, APIs return values arg^f, global variables g, literals l, and memory regions accessible from these values $v.\texttt{field}$ [62, 67]. Meanwhile, a use $u \in U$ abstracts away the propagation of value through intermediate statements, focusing on ultimate usages, including being used as outgoing data of interfaces or involved in sensitive operations, e.g., $deref$, div. The outgoing data contains variables passed to APIs as arguments arg^f, those returned by interface ret^i, and those assigned to global variables g, which would be subsequently used by other entities.

```
1  struct vb2_ops cx23885_qops = {
2    /* Interface: function pointer */
3    .buf_prepare = buffer_prepare,
4  };
5  int buffer_prepare(struct vb2_buffer *vb) {
6    /* Incorrect error code leads to npd */
7  - cx23885_vbibuffer(&buf->risc);
8  + return cx23885_vbibuffer(&buf->risc);
9  }
10 int cx23885_vbibuffer(struct cx23885_riscmem *risc) {
11   /* Uniform API invocation */
12   risc->cpu = dma_alloc_coherent(...);
13   if (risc->cpu == NULL) return -ENOMEM;
14 }
```

Figure 3. A security patch that introduces a new value-flow path from line 13 to line 8.

The path relation R is defined as composite logical constructs that can be expressed through a combination of two basic relations. First, the reachability relation $v \overset{c}{\hookrightarrow} u$ states that a value v reaches a use u under a specific path condition $c \in C$, defined as a first-order logic formula involving values V. Second, the order precedence relation $u_1 \prec u_2$ constrains the partial order of two use sites in the control flow, allowing us to capture critical positional information. By combining these basic relations using logical operators, i.e., $\wedge, \vee, \neg$, we can create highly expressive specifications that effectively capture various anomalous behaviors within the interface.

4.2 Examples

We provide three concrete security patches as examples to demonstrate the expressiveness of our specifications. The value-flow path changes are also provided visually in Fig. 6. Our core insight is the way value-flow path changes could shed light on the form of specifications.

Example 4.1. Incorrect return value. Fig. 3 fixes the NPD in Fig. 1 by correcting the return value at line 8.

Value-flow Changes. In Fig. 6(a), the code changes introduce a new data flow edge, i.e., from the returns of function `cx23885_vbibuffer@7/8` to the returns of function `buffer_prepare@8`, making `-ENOMEM` reach the returns of `buffer_prepare` across function boundaries. The new reachability relation is held under the formula φ_1 over API return values. We use Spec. 4.1 to describe such behavior:

Spec 4.1. $\forall\, v : v \overset{c}{\hookrightarrow} u$, where: (1) v = `-ENOMEM`, (2) $u = ret^{\texttt{buf\_prepare}}$, and (3) $c = ret^{\texttt{dma\_alloc\_cohenrent}} ==$ `NULL`.

Explanation. This specification exemplifies a typical reachability relation between a value and its use. It indicates that the failure of API `dma_alloc_coherent` should be propagated back to the invoker of interface `buf_prepare` via error code `-ENOMEM`. The absence of such a value-flow path implies specification violations. The specification differs from the value-flow changes in three aspects: 1) intermediate statements specific to patched codes are omitted; 2) the use is abstracted to $ret^{\texttt{buf\_prepare}}$ rather than the returns of specific interface implementation; 3) quantifier is associated with regulated interaction data v, while that for u is disregarded since an interface only has a single return.

```
1  struct i2c_algorithm smbus_algorithm = {
2     /* Interface: function pointer */
3     .smbus_xfer = xfer_emulated,
4  };
5  int xfer_emulated(int size, union smbus_data *data) {
6    switch (size) {
7    case I2C_SMBUS_I2C_BLOCK_DATA:
8   +  if (data->len <= MAX)
9        for (i = 1; i <= data->len; i++)
10         msg[0].buf[i] = data->block[i];  /* Out-of-bound bug */
11     }
12 }
```

Figure 4. A security patch that modifies the condition of the value-flow path from line 5 to line 10.

```
1  struct platform_driver telem_driver = {
2     /* Interface: function pointer*/
3     .remove = telem_remove,
4  }
5  struct ida telem_ida;
6  int telem_remove(struct platform_device *pdev) {
7   -  put_device(&pdev->dev);
8      /* Use-after-free bug */
9     ida_free(&telem_ida, MINOR(pdev->dev.devt));
10  +  put_device(&pdev->dev); /* Uniform API invocation */
11 }
```

Figure 5. A security patch that alters the execution orders of two use sites of `pdev->dev`.

Example 4.2. Missing checking on parameter. Fig. 4 showcases a security patch that fixes out-of-bound access in line 10 by adding sanity checks on `data->len`.

Value-flow Changes. As illustrated in Fig. 6(b), the code changes alter the condition that guards the path from parameter of `xfer_emulated`, i.e., `data->block@5` to the dereference site at line 10. A new logic formula φ_3 regarding parameter `data->len` is added to the condition while the path itself stays the same. The specification for it is listed:

Spec 4.2. $\forall v : \nexists u : v \overset{c}{\hookrightarrow} u$, where: (1) $v = arg_2^{\texttt{smbus\_xfer}}$`.block`, (2) $u = deref$, and (3) $c = arg_2^{\texttt{smbus\_xfer}}$`.len` $>$ `MAX`.

Explanation. Different from the first specification, this specification requests the absence of value-flow paths from the `block` field of arg_2 of interface `smbus_xfer` to pointer dereference when the `len` field of arg_2 is larger than `MAX`. Likewise, the specification abstracts variable `data` to arg_2 and attach quantifier on v and u. Besides, the specification does not incorporate φ_2 and φ_4, but retains φ_3. Moreover, we negate φ_3, plus the non-existent quantifier $\nexists$, to collaboratively represent the rejection of invalid value-flow paths.

Example 4.3. Incorrect interface usage order. We finally give a security patch in Fig. 5 to explain how patches could merely change the relative positions of statements in lines 7 and 9 to prevent use-after-free bug.

Value-flow Changes. In Fig. 6(c), the value-flow path that is originally from `pdev->dev@6` to `put_device@7` now flows to `put_device@10`, and notably, no extra path or path condition changes are discovered. To capture the impact of subtle code change, a novel dimension, i.e., use site order Ω, is incorporated. For the two use sites of the tracked variable `pdev->dev`, the dereference site at line 9 was executed after

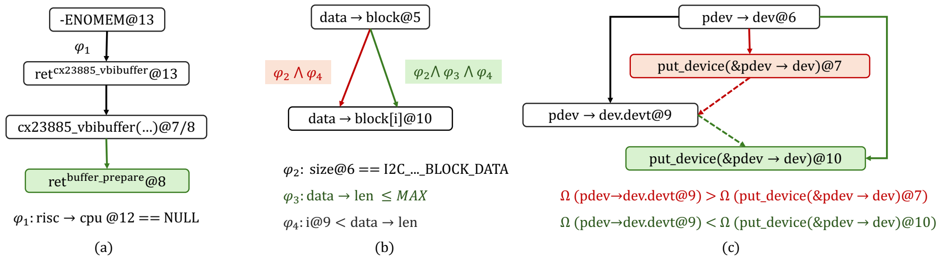

-ENOMEM@13
φ1
ret^cx23885_vbibuffer@13
cx23885_vbibuffer(...)@7/8
ret^buffer_prepare@8
φ1: risc → cpu @12 == NULL
(a)
data → block@5
φ2 ∧ φ4
φ2∧ φ3 ∧ φ4
data → block[i]@10
φ2: size@6 == I2C_..._BLOCK_DATA
φ3: data → len ≤ MAX
φ4: i@9 < data → len
(b)
pdev → dev@6
put_device(&pdev → dev)@7
pdev → dev.devt@9
put_device(&pdev → dev)@10
Ω (pdev→dev.devt@9) > Ω (put_device(&pdev → dev)@7)
Ω (pdev→dev.devt@9) < Ω (put_device(&pdev → dev)@10)
(c)

Figure 6. The value-flow path changes induced code changes in three patch examples. Each node represents a variable/use site, plus the line number. The solid edges imply the data dependence and the dashed arrow denotes the flow order edge. Removals (additions) are annotated in red (green).

`put_device` in pre-patch code, but their orders are reversed thereafter. The specification is listed below:

Spec 4.3. $\nexists u_1, u_2 : (v \hookrightarrow u_1) \wedge (v \hookrightarrow u_2) \wedge (u_2 \prec u_1)$, where: (1) $v = arg_1^{\texttt{remove}}$.`dev`, (2) $u_1 = deref$, and (3) $u_2 = arg_1^{\texttt{put\_device}}$.

Explanation. In contrast to earlier specifications that solely encompass reachability relations, the order precedence relation $u_2 \prec u_1$ emphasizes the desired execution orders of multiple use sites for the same critical interaction data. Specifically, for arg_1 of interface `remove`, there must not exist a dereference site u_1 that occurs after the same variable is passed to API `put_device` as the first argument. There is no quantifier for v since the argument for one interface could be uniquely identified with an index.

5 Seal in a Nutshell

We have demonstrated the benefits of leveraging value-flow properties. Next, we explain how Seal automates specification inference and violation detection in a top-down manner.

Workflow of Seal. As shown in Fig. 7, the input of Seal is the code changes in security patches, where patch descriptions are excluded. Seal would first analyze data-, control- and flow- dependence for pre- and post-patch codes to construct the two versions of dependence graphs [62, 68, 89] (stage ❶). Later, we discern graph differences in the granularity of value-flow paths to perceive dependence changes in three dimensions (stage ❷). Afterward, path changes are generalized to interface specifications (stage ❸). Finally, codes that implement or use the same interface should conform to the same constraint and thus are detection targets. We perform flow-, context-, path-, and field-sensitive value-flow path searching on bug regions to judge violations (stage ❹).

Running Example. We take the security patch in Fig. 5 as a concrete example to detail our workflow.

Step 1: Slicing on PDG. Seal first flattens the pre-patch and post-patch graph structures to paths by performing slicing from interaction data, including parameter `pdev`, global variable `telem_ida`, fields `pdev->dev` and `pdev->dev.devt`. In pre-patch PDG, except for two paths from `pdev->dev@6` to the API invocation site `put_device@7` (#1a) and deref site at line 9 (#2), another two paths from `telem_ida` (#3) and `pdev->dev.devt@6` (#4) to first and second argument of API invocation site `ida_free@9` are also collected. After applying the patch, Seal observes a new path from `pdev->dev@6` to `put_device@10` (#1b).

Step 2: Path Comparison. Seal compares collected paths {#1a, #2, #3, #4} and {#1b, #2, #3, #4} in three aspects: presence in pre-/post- PDG, path conditions, and orders of comparable use sites. The four paths are unchanged for the first two dimensions, particularly for #1a and #1b, since the statements inside paths are identical despite different line numbers, and the path conditions are always `true`. For the last dimension, the use sites in paths #3 and #4 are not comparable with those in paths #1a/#1b/#2 owing to different variables being tracked. Specifically, the type of tracked variable in #3 is integer, therefore, it is passed by value and cannot be modified in API via reference. `telem_ida` in #4 is not alias with `pdev` and we cannot assume one API could manipulate arbitrary memory. As a result, the orders of use sites in #3 and #4 are unchanged. Instead, we notice that $\Omega(9) > \Omega(7)$ in pre-patch code, but $\Omega(9) < \Omega(10)$, making paths #1a, #1b and #2 the ingredient to mine specifications.

Step 3: Specification Abstraction. The paths #1a, #1b, and #2 are abstracted in this step to produce specifications. First, the variable `pdev` specific to `telem_remove` is mapped to the domain of formulation $arg_1^{\texttt{remove}}$ when forming basic reachability relations $v \hookrightarrow u_1$ and $v \hookrightarrow u_2$ and order relation $\prec$. Since order relations between two use sites are only meaningful when they use the same data, Seal applies a conjunction of these three fundamental relations. Second, for the quantifier, the removal of order relation $u_2 \prec u_1$ indicates the quantifier of the above logical conjunction could only be $\exists$ or $\nexists$. Finally, $\exists$ is purged since no other invocations to API `put_device` and dereference sites are found in the patched code. We highlight that the "free" semantic of API `put_device` is not needed in the inference process.

Step 4: Bug Detection. Finally, Seal applies the specifications to other implementations of interface `remove`. The value and use components within the specification are first instantiated inside code regions under inspection. If no corresponding values or uses are found, Seal ceases the analysis of the current implementation and switches to the next one. Otherwise, Seal starts to check the reachability relation by searching for value-flow paths between variables and use

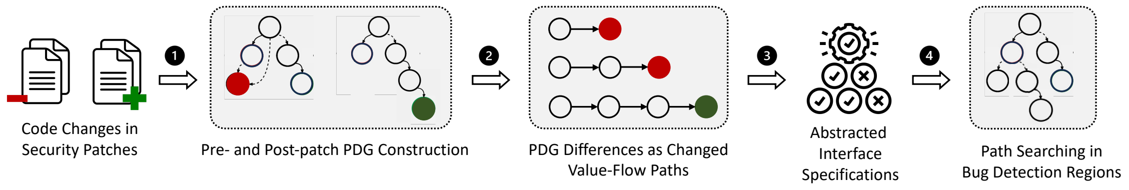

Code Changes in Security Patches
Pre- and Post-patch PDG Construction
PDG Differences as Changed Value-Flow Paths
Abstracted Interface Specifications
Path Searching in Bug Detection Regions

Figure 7. Workflow of Seal.

sites of interest. Once obtained, Seal turns to validate the orders of use sites to determine the presence of violations.

Remark. Seal is agnostic to API semantics, thus choosing to conservatively apply the above specification to other implementations of function pointer `remove` rather than arbitrary codes for the sake of precision and scalability. The rationale is multiple implementations of the same function pointer share the same context and their interaction data is processed uniformly. For instance, `put_device` is designed to decrement the refcount of incoming parameters, and will release it if the refcount reaches zero. For interface `remove` that is typically invoked when unloading a device, the refcount of $arg_1^{\texttt{remove}}$`.dev` is presumably 1. However, `put_device` could be executed before the dereference site in other places if the refcount is >1. Nevertheless, our formulation could be flexibly adjusted by not specifying the applicable function pointers if no relevant elements are involved to increase recall. One example is that the return values of API `kmalloc` should not be dereferenced when it is `NULL` no matter the surrounding codes.

6 Technical Design

We start the technical details of Seal with our customized program dependence graph (PDG) (§ 6.1). Then, we elaborate on several critical technical choices when abstracting specifications from changed value-flow paths (§ 6.2 and § 6.3) and examining specifications for bug detection (§ 6.4).

6.1 PDG Construction

Def. 6.1 defines PDG that effectively captures data-, control-, and flow dependencies between program elements, enabling precise semantic understanding of security patches.

Definition 6.1 (Program Dependence Graph)**.** For a program, its PDG is a 4-tuple $\mathcal{G} := (\mathcal{V}, \mathcal{E}_d, \mathcal{E}_c, \mathcal{E}_o)$, where:

- $\mathcal{V}$ is the set of nodes. Each node $v \in \mathcal{V}$ is a statement or, equivalently, the variable defined by the statement.
- $\mathcal{E}_d \subseteq \mathcal{V} \times \mathcal{V}$ is the set of data-dependence edges. Edge $(v_1, v_2) \in \mathcal{E}_d$ means the value of v_1 is propagated to v_2 1) directly via assignment, 2) indirectly via pointer dereferences, 3) inter-procedurally via actual/formal parameters, return values/receivers during function invocation.
- $\mathcal{E}_c \subseteq \mathcal{V} \times \mathcal{V}$ is the set of control-dependence edges. Edge $(v_1, v_2) \in \mathcal{E}_c$ means that the execution of statement v_2 is determined by the outcome of v_1.
- $\mathcal{E}_o \subseteq \mathcal{V} \times \mathcal{V}$ is the set of control flow edges. Edge $(v_1, v_2) \in \mathcal{E}_o$ indicates that if a concrete execution goes through both v_1 and v_2, v_1 must be executed before v_2.

Both graph differences and bug detection are conducted in the unit of value-flow paths, defined in Def. 6.2. Each value-flow path p describes how a specific variable is propagated and used, which is collected by slicing [80] on PDG along data-dependence edges. We define $v_1 \rightsquigarrow v_2$ if a value-flow path exists from v_1 to v_2. Moreover, each value-flow path is accompanied by several critical information for us to perceive its changes.

Definition 6.2 (Value-flow Path)**.** Given a PDG $\mathcal{G}$, a value-flow path is defined as $p = (v_1, \ldots, v_n)$, where $(v_{i-1}, v_i) \in \mathcal{E}_d$ for $2 \leq i \leq n$. For each p, we have:

- v_1 and v_n represent the source and sink of p.
- $\Psi(p)$ maps p to logical formulas over $\mathcal{V}$ to denote that the path occurs only when constraints are satisfied.
- $\Omega : \mathcal{V} \mapsto \mathbb{N}$ maps each statement in p a partial order where $\Omega(s_1) < \Omega(s_2)$ means s_1 is executed prior to s_2.

Path condition $\Psi(p)$ is computed by recursively traversing control and data dependence edges on PDG, following existing quasi-path-sensitive analysis [62, 71]. $\Omega(v)$ is calculated by performing topological sorting over statements following the control flow edges $\mathcal{E}_o$. In the following sections, the value-flow paths are inter-procedural by default. We use Ψ_- and Ψ_+ to distinguish path conditions collected in pre- and post-patch PDGs when necessary, so as Ω_- and Ω_+. For simplification, we use v to indicate a value on PDG for shorthand and omit the statement defining the value.

6.2 PDG Differentiation

Taking security patches and pre-/post-patch PDGs as inputs, this phase computes PDG differences as changed inter-procedural value-flow paths P_{pre} and P_{post}, serving as the ingredients for specification inference.

Specifically, this stage seeks to identify value-flow paths associated with interaction data and whose data, control, and flow dependencies are altered by patches. In essence, the collection process is conducted via forward and backward slicings from the slicing criterions. We search paths inter-procedurally since interaction data could be propagated and used across functions, as exemplified in § 3.2.

6.2.1 Slicing Criterion. Conceptually, we consider a PDG node as the slicing criterion if either the node itself or one of its incoming/outgoing dependence edge is changed. The set of criterions gradually expands as more PDG nodes with changed dependencies are discovered during slicing.

- First, PDG nodes corresponding to changed source codes are considered as slicing criterions since they would change data dependence relationships. For example, line 8 in Fig. 4 introduces a new comparison statement, whose value-flow paths should be analyzed to figure out whether its value is determined by interaction data.
- Second, if the control dependence edge of a PDG node is altered, the node is considered to be slicing criterion since the path conditions of value-flow paths passing the nodes are modified. Still in Fig. 4, the execution of the array access statement in line 10 additionally depends on the if-condition in line 8 after applying the patch.
- Finally, if the control flow dependence of a PDG node is changed, the node would be sliced. For instance, the flow dependences of nodes in line 8 of Fig. 5 are modified, thus we collect their paths to analyze the order relations with changed lines 7 and 10.

6.2.2 Slicing Termination.
Once starting the forward and backward traversal from picked slicing criterions, we terminate it when encountering interaction data or leaf nodes that have no further data-dependence edge. Eventually, the sources of our collected paths are input data from interfaces and sinks are output data or sensitive operations. These sources and sinks are elements in our formulations and have attributes for us to abstract specifications. Instead, if a path ends in local variables that are irrelevant to interaction data, we discard them. After gathering all changed paths, we calculate their path conditions and record the orders of internal use sites by referring to Def. 6.2. Likewise, path conditions are additionally processed in the same principles to only retain conditions over interaction data. Specifically, we validate whether each variable in constraint Ψ depends on interaction data by traversing along data dependence backward.

6.2.3 Performance Optimization via Memorization.
The above procedure would incur significant overhead due to inter-procedural searching, let alone computing path conditions. To address the problem, we determined to first perform slicing within patched functions, and then memorize visited value-flow paths as summaries to avoid redundant searching when extending the paths into inter-procedural ones.

Specifically, we first terminates slicing when reaching function boundaries, such as arguments, return values, or function calls, although they are not interfaces, e.g., driver-specific function `cx23885_vbibuffer` in Fig. 3. Later on, starting from these boundaries, we recursively perform slicing inside the callers and callees of the patched functions and memorize slicings related to function arguments, and return values to summaries. When the same function arguments and return values are met in subsequent traversal, these summaries could be directly concatenated to the current slicing.

Algorithm 1: Value-flow Path Classification.

```
1 Function Classify($P_{pre}$, $P_{post}$):
2     $P_- \leftarrow P_{pre} \setminus P_{post}$; $P_+ \leftarrow P_{post} \setminus P_{pre}$;
3     $P_\Psi \leftarrow \emptyset$; $P_\Omega \leftarrow \emptyset$;
4     for $p \in P_{pre} \cup P_{post}$ do
5         if $\Psi_+(p) \neq \Psi_-(p)$ then $P_\Psi \leftarrow P_\Psi \cup \{p\}$ ;
6         else $P_\Omega \leftarrow P_\Omega \cup \{p\}$ ;
```

6.3 Specification Extraction
A single value-flow path is meaningless, instead, change matters to understand the correct ways to implement or use interfaces. This stage classifies P_{pre} and P_{post}, into four categories and performs different deduction strategies on each category to infer specifications.

6.3.1 Path Classification.
Alg. 1 segregates P_{pre} and P_{post} into four distinct sets to ascribe divergences. Two paths are identical if they are in the same length, share identical nodes and internal data dependence edges. The algorithm first performs a minus operation on P_{pre} and P_{post} to identify paths that lack a counterpart in another PDG (line 2) and stores them into P_- and P_+. For the remainings, we evaluate the equivalences of path conditions collected in two versions of PDGs (line 5) by examining the consistencies of invovling variables and their satisfying sets. Finally, in case path conditions are identical, the path differences lie in the orders of internal use sites (line 6) and we put these paths into P_Ω.

6.3.2 Specification Abstraction.
We move forward to process paths in four sets to deduce specifications Q in Alg. 2. At a high level, we use the relations among variables, i.e., $R^{\mathcal{V}}$, as the intermediate structure to assist the transitions from path differences to specifications Q. In essence, $R^{\mathcal{V}}$ is an instantiation of relation R in Fig. 2 by replacing elements in V and U with variables in patched codes.

Alg. 2 collects reachability relation $\rightsquigarrow$ from every single path, while the condition and order relation are collected by concerning paths in P_Ψ and P_Ω. Specifically, in lines 3-6, for each path p in P_- and P_+, we extract its source v_1, sink v_n and path condition $\Psi(p)$ to form a reachability relation. The reachability relations in P_- are not expected, whereas the converse is true for P_+. Then, for each path p in P_Ψ, we use delta constraints Ψ_δ (lines 7-9) to form reachability relation, which calculates conditions that only appear in pre- or post-patch PDGs. For instance, in Fig. 4, the condition on `size` is removed when computing delta since it's not changed, instead, we retain the condition on `data->len`. Finally, in lines 11-13 and line 19, for paths in P_Ω, we group their sink statements into mapping $\mathbb{M} : \mathcal{V} \mapsto 2^{\mathcal{V}}$ by source statement v_1. Given source v_1, $\mathbb{M}$ maintains all use sites of v_1 whose orders are comparable. In lines 13-19, upon appending, we iterate over statements $s \in \mathbb{M}[v_1]$ and identify whether the relative positions between current sink v_n and s are inconsistent in pre-/post-patch PDGs. The order relations in pre-patch PDG

Algorithm 2: Specification Deduction.

```
Function Deduction($P_-$, $P_+$, $P_\Psi$, $P_\Omega$):
    $R^{\mathcal{V}} \leftarrow \emptyset$; initialize map $\mathbb{M}$;
    for $p = \{v_1, \ldots, v_n\} \in P_-$ do
        $R^{\mathcal{V}} \leftarrow R^{\mathcal{V}} \cup \{\neg (v_1 \rightsquigarrow v_n \ under \ \Psi_-(p))\}$;
    for $p = \{v_1, \ldots, v_n\} \in P_+$ do
        $R^{\mathcal{V}} \leftarrow R^{\mathcal{V}} \cup \{(v_1 \rightsquigarrow v_n \ under \ \Psi_+(p))\}$;
    for $p = \{v_1, \ldots, v_n\} \in P_\Psi$ do
        $\Psi_\delta = \Psi_-(p) \land \neg \Psi_+(p)$;
        $R^{\mathcal{V}} \leftarrow R^{\mathcal{V}} \cup \{\neg (v_1 \rightsquigarrow v_n \ under \ \Psi_\delta)\}$;
    for $p = \{v_1, \ldots, v_n\} \in P_\Omega$ do
        $r_1 \leftarrow v_1 \rightsquigarrow v_n \ under \ \Psi(p)$;
        if $v_1 \notin \mathbb{M}$ then $\mathbb{M}[v_1] \leftarrow \emptyset$;
        for $s \in \mathbb{M}[v_1]$ do
            $r_2 \leftarrow v_1 \rightsquigarrow s \ under \ \Psi(p' = \{v_1, \ldots, s\})$;
            if $\Omega_+(s) > \Omega_+(v_n) \land \Omega_-(s) < \Omega_-(v_n)$ then
                $R^{\mathcal{V}} \leftarrow R^{\mathcal{V}} \cup \{\neg (r_1 \land r_2 \land (\Omega(s) < \Omega(v_n))\}$;
            if $\Omega_+(s) < \Omega_+(v_n) \land \Omega_-(s) > \Omega_-(v_n)$ then
                $R^{\mathcal{V}} \leftarrow R^{\mathcal{V}} \cup \{\neg (r_1 \land r_2 \land (\Omega(v_n) < \Omega(s))\}$;
        $\mathbb{M}[v_1] \leftarrow \mathbb{M}[v_1] \cup \{v_n\}$;
    $Q \leftarrow$ inferQuantifier($R^{\mathcal{V}}$);        ▷ Explained in § 6.3.3
```

is not expected. We finally deduce specifications Q from $R^{\mathcal{V}}$ with subprocedure `inferQuantifier` (line 20).

6.3.3 Domain Mapping. $R^{\mathcal{V}}$ collected in Alg. 2 contains the reachability and order relations among variables in the patched codes. To form specifications, we need to abstract program variables in domain $\mathcal{V}$ to elements in formulations and deduce the quantifiers for collected relations, which is conducted by subprocedure `inferQuantifier`.

First, we use a mapping $\mathbb{A} : \mathcal{V} \mapsto V \cup U$ to abstract program variables $\mathcal{V}$ to V and U, thereby deriving R from $R^{\mathcal{V}}$. The mapping is many-to-one, for instance, multiple dereference sites in programs could be mapped to $deref$. For reachability relation, the source v_1 is mapped to V, indicating regulated interaction data. The variables inside path conditions Ψ are mapped to V similarly to form condition C. The sink v_n is mapped to U, representing the critical use to check. The order relation between two statements in $\mathcal{V}$ is kept when transforming them to U, e.g., $\Omega(s_1) < \Omega(s_2)$ is preserved by $u_1 \prec u_2$ where $\mathbb{A}[s_1] = u_1$ and $\mathbb{A}[s_2] = u_2$.

Second, the subprocedure infers quantifiers for relations R. Our insight is the way relations change implies the possible quantifiers. The relations in $R^{\mathcal{V}}$ with $\neg$ are removed when applying patches, whose corresponding quantifiers could only be $\exists$ or $\nexists$. Instead, the quantifiers for relations in $R^{\mathcal{V}}$ without the negation operator could be $\exists$ or $\forall$. These quantifiers are further validated by examining the relations among variables that could map to the same value V/use U in patched code. Taking $\neg(v_1 \rightsquigarrow v_n) \in R^{\mathcal{V}}$ as an example, after abstraction, we obtain $\mathbb{A}(v_1) \hookrightarrow \mathbb{A}(v_n) \in R$ whose quantifier is $\exists$ or $\nexists$. Then, we validate whether the reachability relation among other variables that also map to $\mathbb{A}(v_1)$ or $\mathbb{A}(v_n)$ hold. If not, we conclude the quantifier is $\nexists$.

6.4 Path-Sensitive Bug Detection

Given the summarized specifications, the core of bug detection is a graph reachability problem, whose goal is to gather realizable value-flow paths in bug detection regions and examine if they conform to specifications Q.

6.4.1 Path Searching. Specifically, we consider the bug detection regions to be other implementations of the same function pointer, determined via indirect call reasoning [22, 50], or other usages of the same API if no elements related to function pointers are involved. For each region, we first locate the set of sources $\mathcal{V}_{src}$ and sink $\mathcal{V}_{sink}$ statements, by mapping V and U to PDG nodes $\mathcal{V}$, which is an inverse of mapping $\mathbb{A}$. If $\mathcal{V}_{src}$ or $\mathcal{V}_{sink}$ is empty, we cease bug detection. Otherwise, for each source-sink pair, we perform depth-first searching on PDGs in a flow-, context-, path-sensitive, inter-procedural manner. During traversal, path conditions are collected and evaluated simultaneously to pause infeasible paths that involve contradictory conditions timely.

To save computation costs, we enforce the same memorization strategies to cache intra-procedural value-flow paths as summaries to avoid redundant searching. Unlike the slicing to collect changed paths, the inter-procedural analysis here is in a bottom-up fashion [63], where before analyzing a function, all its callee functions are analyzed and their paths are memorized as function summaries. The final value-flow paths are obtained via summary inlining that stitches paths and takes conjunction of path conditions.

6.4.2 Path Examination. We identify and report bugs if the collected value-flow paths contravene any specified constraints. Initially, for each path p, we assess the consistency between $\Psi(p)$ and given condition $c \in C$. This is achieved by first inversely mapping the interaction data within condition c to variables in target regions using the inverse of mapping $\mathbb{A}$, followed by an evaluation of the satisfying sets. If not, we further employ a relaxed strategy to accommodate code differences by verifying whether critical interaction data in condition c is present in $\Psi(p)$. Subsequently, we evaluate the consistency of order relations for use sites that are reachable from the same source. Finally, we analyze the quantifiers over all collected realizable value-flow paths.

7 Implementation

Seal is developed on top of LLVM [42], where the programs are in SSA form [29]. The PDGs are constructed with sparse value-flow analysis [62, 67] and logical satisfiability is determined by Z3 solver [30]. More details are provided below.

LLVM Bitcode Generation. Applying a security patch involves two versions of codes. To facilitate extensive node and edge comparisons, we link two versions of codes into a single bitcode. Specifically, we rename global variables and functions in pre-/post-patch codes to distinguish them and prevent linkage errors. Besides, we enable the option "-g" to preserve line numbers of PDG nodes and utilize SIRO [93] to overcome the IR version incompatibility between our implementation and the compilation chain of Linux.

Demand-driven PDG Generation. The cost of PDG generation for the entire program is excessive, and such cost is further amplified when calculating differences. Thus, we choose to generate PDGs on demand for the sake of efficiency. Specifically, in PDG differentiation, we only generate PDGs for patch-related functions, which are analyzed by performing lightweight call graph analysis to identify the callers and callees of patched functions until we meet interfaces. Likewise, we solely generate PDGs for delineated bug detection regions when checking violations.

Value-flow Analysis. Our value-flow analysis is context-, field-, flow- and path-sensitive. We follow existing efforts [62, 89] to reason alias and assume APIs could read/write passing pointer parameters and accessible fields. The structure fields are distinguished by the bytes offsets from the base pointer. Context-sensitivity is reached by applying CFL reachability [24, 60] during PDG slicing, and by function cloning [81] during bottom-up bug detection. Indirect calls are resolved by type analysis [22] when determining bug detection regions. Note that our slicing does not cross the boundary of function pointers to ensure scalability.

Bug Report. SEAL provides user-friendly bug reports to ease the burden of bug confirmations and fixes. The bug reports contain buggy value-flow paths with line numbers attached, inferred specifications, and the original patch, which shed light on the bug-fixing suggestions.

8 Evaluation

To quantify the effectiveness and efficiency of SEAL, we propose the following four research questions:

- **RQ1**: How effective are inferred specifications in finding previously unknown bugs in Linux (§ 8.1)?
- **RQ2:** What are the characteristics of interaction data behaviors being regulated and their violations? (§ 8.2)?
- **RQ3**: How does SEAL compare against patch-based and deviation-based approaches (§ 8.3)?
- **RQ4**: How efficient does SEAL in inferring specifications and detecting violations (§ 8.4)?

Dataset. SEAL targets Linux v6.2 (commit `c9c3395d`), the latest version at the time of the experiments. To collect inputs, we enumerated the historical patches applied to Linux v6.2 and followed existing approaches [47] to search keywords over patch descriptions to identify commits for bug fixes. These keywords pertain to typical terms used to describe 15 bug types that SEAL proficiently handles, such as "oob" for out-of-bounds. Finally, in total of 12,571 security patches were collected. The source code of SEAL is available[1].

Environment. All experiments are carried out on a single 64-bit server running Ubuntu 20.04 LTS and equipped with 64

[1] https://github.com/harperchen/SEAL.git

Table 1. 45 bug samples found by SEAL. The S, C, A in column **Status** represent Submitted, Confirmed, and Applied.

SubSystem (Location)	Buggy function	Type	Status
drivers/media/usb	rtl28xxu_i2c_xfer	NPD	A
drivers/media/usb	gl861_i2c_master_xfer	NPD	A
drivers/media/usb	dw2102_i2c_transfer	NPD	A
drivers/media/usb	ce6230_i2c_master_xfer	NPD	A
drivers/video/fbdev	tgafb_check_var	DbZ	A
drivers/video/fbdev	nvidiafb_check_var	DbZ	A
drivers/video/fbdev	au1200fb_fb_check_var	DbZ	A
drivers/staging	ks_wlan_set_encode_ext	OOB	A
drivers/media/pci	tw68_buf_prepare	NPD	A
drivers/media/pci	buffer_prepare	NPD	A
drivers/i2c/busses	xgene_slimpro_i2c_xfer	OOB	A
drivers/regulator	stm32_adc_probe	NPD	C
drivers/firmware	meson_sm_probe	NPD	A
drivers/dma	mv_xor_probe	NPD	S
drivers/bus	weim_parse_dt	NPD	S
drivers/video/fbdev	au1200fb_drv_probe	Wrong EC	A
drivers/spi	tegra_slink_probe	Wrong EC	S
drivers/sound/soc	rt5665_i2c_probe	MemLeak	A
drivers/tty	asc_init_port	NPD	C
drivers/spi	tegra_sflash_probe	Wrong EC	S
drivers/mmc/host	spmmc_drv_probe	MemLeak	A
drivers/net/wireless	rtw89_debug_priv_send_h2c_set	Wrong EC	A
drivers/net/wireless	rtw_debugfs_set_fix_rate	Wrong EC	A
drivers/net/wireless	rtl_debugfs_set_write_reg	Wrong EC	A
drivers/media/usb	opera1_read_mac_address	Uninit Val	A
drivers/media/usb	su3000_read_mac_address	Uninit Val	S
drivers/usb	gfs_bind	Wrong EC	A
drivers/media/i2c	hi846_init_controls	MemLeak	A
drivers/platform	vb2ops_venc_queue_setup	OOB	A
drivers/platform	viacam_probe	UAF	A
drivers/media/pci	netup_unidvb_initdev	NPD	A
drivers/md	multipath_remove_disk	OOB	A
drivers/md	raid1_remove_disk	OOB	A
drivers/ata	ahci_platform_get_resources	MemLeak	S
drivers/iommu	mtk_iommu_of_xlate	MemLeak	S
drivers/dma	lpc18xx_dmamux_reserve	MemLeak	S
drivers/edac	amd8131_probe	MemLeak	S
drivers/media/usb	go7007_register_encoder	NPD	S
drivers/usb/dwc3	dwc3_imx8mp_probe	MemLeak	S
drivers/firewire	fwnet_finish_incoming_packet	UAF	A
fs/ext4	ext4_parse_param	NPD	C
fs/quota	dquot_init	MemLeak	S
net/sched	tcf_gate_cleanup	NPD	S
net/hsr	prp_get_untagged_frame	NPD	S
core/mm	shmem_parse_one	NPD	S

Note: the complete bug list is available at https://harperchen.github.io/bugs.html.

Intel(R) Xeon(R) Gold 6226R CPU cores running at 2.90GHz and 252GB of memory.

8.1 Effectiveness of SEAL

Confirmed and Fixed Bugs. SEAL generated 232 bug reports using inferred specifications. All bug reports are manually validated using informative value-flow paths and original patches, relieving us to investigate all reports in two days and confirm 167 true bugs, achieving a precision of 71.9%. We sampled 45 true bugs in Tab. 1 due to the space limits. Among 167 true bugs, 146 of them are driver bugs, 7 for the filesystem, 13 for the network, and 1 for the core subsystem. We have actively reported bugs to kernel maintainers and prepared patches for bug fixing. Upon acceptable, 95 bugs and corresponding specifications were confirmed, and 56

Table 2. Bug types and root causes of reported bugs

Bug types	Prop	Causes	CWE ID
NULL Ptr Deref (NPD)	31.0%	① - ④	CWE-478
Memory/Resource Leak	23.7%	③	CWE-401/402
Wrong Error Codes	19.8%	②, ③	CWE-393
Out of Bounds (OOB)	10.3%	①	CWE-125/787
Use After Free/Double Free	9.2%	②, ④	CWE-415/416
Divide by Zero (DbZ)	4.3%	①	CWE-369
Uninitialized Value	1.7%	②	CWE-456/457

Note: ① incorrect/missing checks of interaction data, ② incorrect return values, ③ incorrect/missing error handling of APIs, ④ incorrect usage orders of APIs

bugs were fixed by our patches. Kernel maintainers quickly reviewed, accepted, and applied our patch due to the clear explanations and original patches as examples. In particular, 27 patches got a response within one day. So far, our patches have been backported to stable versions, demonstrating the effectiveness of SEAL in enhancing Linux reliability.

Bug Types and Security Impacts. In Tab. 2, SEAL discovers a wide range of vulnerabilities, most of which belong to the top 25 most dangerous software weaknesses [7], including one assigned CVE-2023-2194. Specifically, 31.0% of bugs are NULL pointer dereferences (NPD) that could crash the system. Memory leaks (MemLeak) and uninitialized values (Uninit Val) can result in sensitive information exposure. Wrong Error Code (Wrong EC) could hurt system reliability. For exploitability, 33.1% of bugs reside in system call handlers, which could be triggered via crafted system call sequences. 5.3% are mistakes in interrupt handlers. All of them are regarded as user-controllable entry points [48, 54]. Importantly, we have manually triggered one NPD bug by slightly changing the PoC of CVE-2023-28328 [4].

Long Latency. We emphasize that bugs found by SEAL are long-latent, which have been hidden for an average of 7.7 years before being identified. As shown in Fig. 8(a), 29% of bugs have a period of more than 10 years. Note that Linux has been regularly scanned by commercial analyzers, such as Coverity [2], Smatch [5], Syzkaller [74], etc. However, none of our found bugs were detected by those industrial tools, providing a strong evidence for the indispensability of SEAL.

8.2 Specification Characteristics

Specification Statistics. SEAL produced 12,322 relations from security patches to regulate interaction data behaviors. Among them, 2,084 reachability relations were summarized from removed value-flow paths P_-, and 5,499 from added ones P_+ (defined in § 6.3). Additionally, 3,757 reachability relations were abstracted from P_Ψ whose changed conditions matter when forming specifications. The remaining 982 order relations came from P_Ω, describing the control flow relationships between use sites. Interestingly, the removed relations are far fewer than the added ones, implying that developers tend to forget to perform necessary operations on interaction data. Similar tendencies are also observed in conditions: critical checkings on interaction data are often missing. We

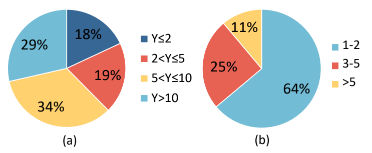


Figure 8. (a) Latent years of reported bugs and (b) the distributions of #violations for specifications (0 is excluded).

```
int wiz_probe(struct platform_device *pdev) {
    serdes = of_get_child_by_name(pdev->dev.of_node, "serdes");
    subnode = of_get_next_child(serdes, NULL);
    ret = of_property_read_u32(subnode, "reg", ...);
    if (ret) {
+       of_node_put(subnode);
        return ret;
    }
}
```

Figure 9. A security patch that adds a API to fix reference leak, from which SEAL produce incorrect specifications.

deduced zero relation for 1,529 security patches since no changed paths related to interaction data are observed.

Specification Correctness. We randomly sampled 1,000 specifications to evaluate the correctness, including 689, 256, and 55 relations focusing on reachability, condition, and order, respectively. Our manual inspection confirmed the precision of specifications is 57.8%. The imprecision preliminary originates from the restrictive information provided by code changes, leading to value-flow paths that are irrelevant to fixed bugs being processed. Particularly, when the values probably depend on multiple APIs or the paths are conditioned by multiple APIs, we would conservatively encompass all of them in specifications. A representative patch is given in Fig. 9. APIs `of_get_next_child` and `of_property_read_u32` are both suspected to require paired API `of_node_put` according to the value-flow paths of `subnode`, whereas only the first is correct. We envision that incorporating more information, e.g., patch descriptions or human intervention, would mitigate the imprecision.

Influence on Bug Detection Precision. All the mined specifications are used to detect bugs. We clarify that incorrect specifications do not hurt bug detection precision harshly since the correct and incorrect specifications do not equally contribute to violation detection. Specifically, we found that out of 232 bug reports, 167 were due to violations of correct specifications, while 53 resulted from false ones. Considering specifications precision is sampled to be 57.8%, we could derive that the probability of violating correct specifications is 2.4%, while the probability of violating incorrect specifications is about 1%. Our manual inspection confirmed that most of the incorrect specifications tend to summarize ad-hoc and abnormal usages of infrequently used interaction data, making it restrictive and cannot be extended.

Specifications and Bug Types. We categorized the root causes of reported bugs into ① to ④ in Tab. 2, which illustrate how different components of our specifications contribute to bug detection. First, bugs with root cause ③ and ④ are reported due to the absence of reachability relation. Second,

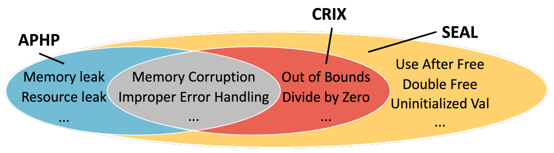


Figure 10. Bug types supported by SEAL and existing efforts.

bugs in ② occur since path conditions of searched paths do not meet the required conditions. Finally, bugs associated with ③ are identified due to inconsistent orders. In conclusion, our specifications in Fig. 2 provide a uniform way to describe a variety of interaction data mishandling patterns. We further presented the distribution of #violations for specifications in Fig. 8(b). While the majority are violated once or twice, 11% of specifications exhibit more than five violations, underscoring the importance of interface specifications.

8.3 Comparison with Existing Tools

We also compared SEAL with existing deviation-based and patch-based work. The precision is measured by manually reviewing bug reports. Since obtaining all bugs in Linux is unrealizable, the false negatives are measured indisputably, where all bugs found by three tools are ground truth. As we discussed below and in Fig. 10, SEAL outperforms existing tools in precision and supported bug types.

Comparison with Patch-based Approaches. We selected APHP [47] as the patch-based tool and performed comparisons on the same set of input patches. APHP is an intra-procedural API post-handling bug detector, which only covers bugs caused by ③ in Tab. 2. The tool generated 28,479 bug reports, 60 of which are confirmed to be true. The low precision primarily stems from incorrect specifications (90.8%) which results from deficiencies in patch processing. Other FPs are mainly caused by imprecise path-sensitive analysis, unknown equivalent post-operations, and intra-procedural design. APHP has a high reliance on patch patterns, once not conformed, would generate incorrect specifications. For FNs, APHP shares 25 memory leak bugs with SEAL but misses others, since it only summarizes post-handling specifications and performs bug detection intra-procedurally. Therefore, APHP would fail to process the patch exemplified in Fig. 3.

Comparison with Deviation-based Approach. We choose CRIX [51] that contrasts conditions of peer slices from the same variables for missing-check bugs. CRIX reported 3,105 missing check bugs, whose reasons fall in ① and ③. We confirmed 44 TPs, including one bug found by SEAL. The reasons for high FPs are three-fold. First, imprecise data flow analysis leads to incomparable slicings being cross-checked. Second, coarse-grained condition modeling causes unaware identical conditions. Finally, empirically determined security checks result in incorrect specifications. The FNs are also attributed to the first two reasons that cause significant slicings being excluded and uncaptured conditions equivalences. SEAL avoids these limitations by modeling conditions with logical formulas and employing precise pointer analysis.

Analysis for SEAL. We confirmed 167 TPs out of 232 reports. The 65 FPs originate from four aspects. First, incorrect specifications bring about 53 FPs. Second, we apply each specification independently during bug detection, which overlooks their equivalences, e.g., APIs `kfree_sensitive` and `kfree`. Third, SEAL examines specifications within each interface individually and ceases bug detection when meeting indirect calls, however, some necessary conditional checks may be placed beyond the current interface. Finally, imprecise indirect call reasoning causes spurious callees to be considered detection targets. For FNs, we missed 35 bugs of APHP due to restrictive bug detection regions. Specifically, when involving function pointer elements, the specifications are only applied in other implementations of the same function pointer. However, blindly detecting bugs in arbitrary contexts would burden the efficiency under inter-procedural design. SEAL failed to find 43 bug of CRIX since no corresponding patches are available.

8.4 Efficiency of SEAL

In the end, we evaluated the execution time of SEAL. The patch processing phase took 30h39m to handle 12,571 security patches sequentially. Note that this is a one-time effort and the summarized specifications can be reused for bug detection. Each patch takes 8.78s approximately, including generating PDGs for two versions of codes, computing graph differences, and deriving specifications.

The bug detection consists of two phases: PDG generation and path searching, which took 5h25m and 1h48m, respectively. This phase searches for realizable value-flow paths guided by summarized specifications and checks consistency. Specifically, SEAL only performs bug detections within delineated code regions, thus avoiding analyzing large calling depths. Notably, the scalability of our technique could be further improved by searching paths in parellel [71] or considering mutual synergy among specifications [61].

9 Discussion

PDG Choices and Applicable Bug Types. SEAL excels at memory-related, missing-check, taint-style, and API misuses bugs, thanks to our general PDG. Other graphs, e.g., code property graph [87], are not precise enough to perceive various dependence changes, failing to fulfill our needs. Nevertheless, the general PDG could be enhanced to convey additional information for more bug types. First, by combining numerical analysis [37], SEAL can determine the three dependencies among variables and their ranges to cope with signedness bugs. Second, SEAL does not support concurrency bugs, e.g., data race, which could be addressed by incorporating auxiliary information, e.g., lock dependence, happens-before relations [18, 23, 84], to enrich current PDG.

Limitation of Specification. The expressiveness of formulation bounds the capability of SEAL. The current formulation describes behaviors involving multiple APIs, such as invocation order, but does not support interactions across multiple function pointers. For instance, some drivers design function pointer `init` to merely allocate resources and return error codes once failed. The upper layer would invoke another interface `fini` to perform the cleanup. In that case, a correct specification should not only claim how to return correct error codes in `init`, but also necessary cleanup operations in `fini` collaboratively. Instead, Seal produces specifications for each interface independently, even if a patch corrects multiple interfaces simultaneously. Although hurting precision, this design offers better generalizability and is easier to reproduce with high scalability since the analysis would not be bothered by deep calling context involving indirect calls.

Future Work of Seal. We believe the precision of Seal could be further enhanced in three aspects. First, security patches could become obsoleted during evolution, even erroneous during manual review. Moreover, prior efforts have also identified the insufficient fix problem [83], where a bug is fixed by a series of follow-up patches. Thus, the evaluations or transformations of patches could offer Seal high-quality inputs, thus are orthogonal to us. Second, we could incorporate more human knowledge, such as large language model (LLM), to process patch descriptions for precise localization of patch-related value-flow paths. Finally, concerning abstracting quantifier formulas from value-flow changes (§ 6.3.3), it's possible to deduce more precise quantifier constraints in high-order form or merge specifications with domain knowledge instead of simply appending.

Suggestions for Kernel Developers. We provide several suggestions gained from the journey of Seal for the Linux community. First, when new APIs or function pointers are introduced, designers are encouraged to clarify their functionalities, appropriate usages, and potential risks of interaction data. Second, for interfaces that lack explanations, kernel maintainers could gradually supplement these specifications when patches that fix the mishandling of interaction data are observed. Specifically, the maintainer could build a dataset of interface specifications, and once new patches are merged, proactively run Seal to expand the dataset and detect the existence of other violations. Some specifications could also be transformed into rigorous type constraints to be enforced at compilation time. Finally, an inexperienced developer, once confused with interfaces, is suggested to learn from other implementations or usages to gain practical insights.

10 Related Work

Specification Inference. By summarizing program semantics and behaviors as rules, specifications allow static analyzers to quickly grasp program attributes and reuse without delving into the code, thereby scaling to the complexity of modern software. Their forms range from pre/post conditions of API usages [26, 53, 59, 82, 91] to various relationships between program constructs to facilitate various analysis, e.g., points-to analysis [20, 75], taint analysis [26, 88], alias analysis [32, 76], and bug detection. Recent decades have witnessed a large body of efforts to derive specifications from implementations [17, 31, 39, 53], usages [47, 59, 82], documentation [21, 52, 76, 97] and large language model (LLM) [44, 49, 90]. Among them, the usages employed for specification inference contain certain code patterns [26, 53, 82, 91], dynamic execution traces [20, 27, 66], the majority from cross-checking [16, 51, 91], etc.

The patch-based approaches [36, 47] we adopt essentially leverage high-quality usage patterns, i.e., anti-examples and fixes, to derive a more accurate specification. Moreover, instead of generating specifications for limited behaviors, our formulation based on value-flow properties (§ 4.1) offers high versatility, as illustrated in § 8.2.

Linux Vulnerability Detection. Bug detection in Linux has attracted great interest in recent years, mainly focusing on conquering domain-specific natures and scalability issues. Among them, dynamic fuzzing [28, 54, 58, 64, 74, 96] establishes various communication channels on top of potential exploitable surfaces, to feed generated or mutated inputs for kernel. For instance, Periscope [64] intercepts the page fault handling mechanism of the Linux kernel to inject mutated hardware data. Static solutions [18, 19, 39, 53, 55, 94] tend to tailor general techniques for certain bug types in Linux, i.e., use-before-initialization [92], missing security or permission check [51, 95], taint-style bugs [55], aiming for precise yet scalable bug detection. Our work complements static kernel bug detection by providing interface specifications to detail correct usages of interaction data.

11 Conclusion

Linux interfaces ease the interactions of various subsystems, but the incomplete calling contexts unexpectedly inflict mishandling of interaction data. This work presents Seal that infers interface specifications in the form of value-flow properties from security patches. We benefit from the expressiveness of value-flow properties to describe the complex behaviors of interaction data. Our approach found 167 unseen vulnerabilities in Linux with high precision. So far, 95 bugs have been confirmed by Linux maintainers with a remarkably fast response time, demonstrating the capability of Seal in mitigating the security risks of Linux.

Acknowledgment

We would like to acknowledge and thank the anonymous reviewers and our paper shepherd, Prof. Jia-Ju Bai, for providing constructive suggestions and insightful feedback. This work is supported by research donations from Huawei, TCL, and Tencent. Bowen Zhang is the corresponding author.

References

[1] 2021. Dicussion with kernel maintainers. https://lore.kernel.org/all/YPgbHMtLQqb1kP0l@ravnborg.org/.

[2] 2023. Coverity static analyzer. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html. Accessed on October 18, 2023.

[3] 2023. Linux Source Tree for Upstream Branch. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/.

[4] 2023. Null pointer dereference flaw reproducer. https://lore.kernel.org/lkml/CAO4mrfcPHB5aQJO=mpqV+p8mPLNg-Fok0gw8gZ=zemAfMGTzMg@mail.gmail.com/.

[5] 2023. Smatch: Static Analyser for C. https://github.com/error27/smatch. Accessed on 18 October 2023.

[6] 2024. BUG_ON and WARN_ON. https://github.com/torvalds/linux/blob/master/Documentation/process/coding-style.rst.

[7] 2024. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/.

[8] 2024. Doxygen: Automated Code Documentation. https://www.doxygen.nl/.

[9] 2024. Dynamic DMA mapping Guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt.

[10] 2024. Eiffel Programing Language. https://www.eiffel.org/documentation.

[11] 2024. Linux Kernel Documentation. https://docs.kernel.org/.

[12] 2024. Overview of the Linux Virtual File System. https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

[13] 2024. Platform Devices and Drivers. https://www.kernel.org/doc/Documentation/driver-model/platform.txt.

[14] 2024. Video2Linux devices: struct vb2_ops. https://www.linuxtv.org/downloads/v4l-dvb-internals/device-drivers/API-struct-vb2-ops.html.

[15] Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd international conference on software engineering*. 746–755.

[16] Mansour Ahmadi, Reza Mirzazade farkhani, Ryan Williams, and Long Lu. 2021. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2025–2040. https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi

[17] Steven Arzt and Eric Bodden. 2016. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*. 725–735.

[18] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 255–268. https://www.usenix.org/conference/atc19/presentation/bai

[19] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe {DMA} Accesses in Device Drivers. In *30th USENIX Security Symposium (USENIX Security 21)*. 1629–1645.

[20] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 678–692.

[21] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 242–253.

[22] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the power of type-based call graph construction by using regional pointer information. In *33nd USENIX Security Symposium (USENIX Security 24)*.

[23] Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. 2023. Place your locks well: understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3727–3744.

[24] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[25] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 480–491.

[26] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 760–774.

[27] Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: mining explicit information flow specifications from concrete executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 129–140.

[28] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069

[29] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[30] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 337–340.

[31] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V Thakur. 2019. Effective error-specification inference via domain-knowledge expansion. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 466–476.

[32] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 745–759.

[33] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *The 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*.

[34] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041

[35] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 72–82. https://doi.org/10.1109/ICSE.2019.00025

[36] Pranav Garg and Srinivasan H. Sengamedu. 2022. Synthesizing code quality rules from examples. *Proc. ACM Program. Lang.* 6, OOPSLA2

(2022), 1757–1787. https://doi.org/10.1145/3563350

[37] Yiyuan Guo, Peisen Yao, and Charles Zhang. 2024. Precise Compositional Buffer Overflow Detection via Heap Disjointness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 63–75.

[38] Liang He, Purui Su, Chao Zhang, Yan Cai, and Jinxin Ma. 2023. One Simple API Can Cause Hundreds of Bugs An Analysis of Refcounting Bugs in All Modern Linux Kernels. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 52–65.

[39] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 345–362.

[40] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. 2019. Detecting error-handling bugs without error specification input. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 213–225.

[41] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 472–482.

[42] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[43] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4211–4228.

[44] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[45] Tuo Li, Jia-Ju Bai, Gui-Dong Han, and Shi-Min Hu. 2024. {LR-Miner}: Static Race Detection in {OS} Kernels by Mining Locking Rules. In *33rd USENIX Security Symposium (USENIX Security 24)*. 6149–6166.

[46] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315.

[47] Miaoqian Lin, Kai Chen, and Yang Xiao. 2023. Detecting API Post-Handling Bugs Using Code and Description in Patches. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3709–3726. https://www.usenix.org/conference/usenixsecurity23/presentation/lin

[48] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. 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*. 1627–1644.

[49] Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. 2024. Generating API Parameter Security Rules with LLM for API Misuse Detection. *arXiv preprint arXiv:2409.09288* (2024).

[50] Kangjie Lu and Hong Hu. 2019. Where Does It Go?: Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1867–1881. https://doi.org/10.1145/3319535.3354244

[51] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1769–1786. https://www.usenix.org/conference/usenixsecurity19/presentation/lu

[52] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. 2020. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1837–1852.

[53] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. 2022. Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1566–1566.

[54] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: fuzzing Linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 404–416.

[55] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. 1007–1024.

[56] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 531–544.

[57] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 361–377. https://doi.org/10.1145/2815400.2815422

[58] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2559–2575. https://www.usenix.org/conference/usenixsecurity20/presentation/peng

[59] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks.. In *NDSS*, Vol. 14. 1125.

[60] Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. 2022. Indexing the extended dyck-cfl reachability for context-sensitive program analysis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1438–1468.

[61] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the extensional scalability problem for value-flow analysis frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 812–823.

[62] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. *SIGPLAN Not.* 53, 4 (jun 2018), 693–706. https://doi.org/10.1145/3296979.3192418

[63] Qingkai Shi and Charles Zhang. 2020. Pipelining bottom-up data flow analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 835–847.

[64] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary.. In *NDSS*.

[65] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*. 112–122.

[66] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 198–209.

[67] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference*

on compiler construction. 265–266.

[68] Yulei Sui and Jingling Xue. 2018. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Transactions on Software Engineering* 46, 8 (2018), 812–835.

[69] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 254–264.

[70] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. 2021. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2471–2488. https://www.usenix.org/conference/usenixsecurity21/presentation/tan

[71] Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Zhang. 2024. Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–33.

[72] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 386–396.

[73] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in C. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 752–762.

[74] Dmitry Vyukov. 2022. Syzkaller: coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[75] Chengpeng Wang, Wenyang Wang, Peisen Yao, Qingkai Shi, Jinguo Zhou, Xiao Xiao, and Charles Zhang. 2023. Anchor: Fast and precise value-flow analysis for containers via memory orientation. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–39.

[76] Chengpeng Wang, Jipeng Zhang, Rongxin Wu, and Charles Zhang. 2024. DAInfer: Inferring API Aliasing Specifications from Library Documentation via Neurosymbolic Optimization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2469–2492.

[77] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2409–2426.

[78] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 163–177. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang

[79] Xiaoke Wang and Lei Zhao. 2023. Apicad: Augmenting api misuse detection through specifications from code and documents. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 245–256.

[80] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[81] John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*. 131–144.

[82] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2041–2058. https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi

[83] Zifan Xie, Ming Wen, Zichao Wei, and Hai Jin. 2024. Unveiling the Characteristics and Impact of Security Patch Evolution. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1094–1106.

[84] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1643–1660.

[85] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 661–678.

[86] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 414–425.

[87] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[88] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.

[89] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 567–592.

[90] Chengfeng Ye, Yuandao Cai, and Charles Zhang. 2024. When Threads Meet Interrupts: Effective Static Detection of {Interrupt-Based} Deadlocks in Linux. In *33rd USENIX Security Symposium (USENIX Security 24)*. 6167–6184.

[91] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISAN: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 363–378.

[92] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: A Precise and Scalable Method to Detect Use-before-Initialization Bugs in Linux Kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 221–232. https://doi.org/10.1145/3368089.3409686

[93] Bowen Zhang, Wei Chen, Peisen Yao, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2024. SIRO: empowering version compatibility in intermediate representations via program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 882–899.

[94] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. 2021. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 811–824.

[95] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. {PeX}: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*. 1205–1220.

[96] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. 2022. Semantic-Informed Driver Fuzzing Without Both the Hardware Devices and the Emulators. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.

[97] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.