# Uncovering the iceberg from the tip: Generating API Specifications for Bug Detection via Specification Propagation Analysis

Miaoqian Lin[1,2] Kai Chen[1,2,⋆] Yi Yang[1,2] Jinghua Liu[1,2]
[1]*Institute of Information Engineering, Chinese Academy of Sciences, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*
{linmiaoqian, chenkai, liujinghua, yangyi}@iie.ac.cn

***Abstract*—Modern software often provides diverse APIs to facilitate development. Certain APIs, when used, can affect variables and require post-handling, such as error checks and resource releases. Developers should adhere to their usage specifications when using these APIs. Failure to do so can cause serious security threats, such as memory corruption and system crashes. Detecting such misuse depends on comprehensive API specifications, as violations of these specifications indicate API misuse. Previous studies have proposed extracting API specifications from various artifacts, including API documentation, usage patterns, and bug patches. However, these artifacts are frequently incomplete or unavailable for many APIs. As a result, the lack of specifications for uncovered APIs causes many false negatives in bug detection.**

**In this paper, we introduce the idea of *API Specification Propagation*, which suggests that API specifications propagate through hierarchical API call chains. In particular, modern software often adopts a hierarchical API design, where high-level APIs build on low-level ones. When high-level APIs wrap low-level ones, they may inherit the corresponding specifications. Based on this idea, we present APISpecGen, which uses known specifications as seeds and performs bidirectional propagation analysis to generate specifications for new APIs. Specifically, given the seed specifications, APISpecGen infers which APIs the specifications might propagate to or originate from. To further generate specifications for the inferred APIs, APISpecGen combines API usage and validates them using data-flow analysis based on the seed specifications. Besides, APISpecGen iteratively uses the generated specifications as new seeds to cover more APIs. For efficient and accurate analysis, APISpecGen focuses only on code relevant to the specifications, ignoring irrelevant semantics. We implemented APISpecGen and evaluated it for specification generation and API misuse detection. With 6 specifications as seeds, APISpecGen generated 7332 specifications. Most of the generated specifications could not be covered by state-of-the-art work due to the quality of their sources. With the generated specifications, APISpecGen detected 186 new bugs in the Linux kernel, 113 of them have been confirmed by the developers, with 8 CVEs assigned.**

⋆ Corresponding Author

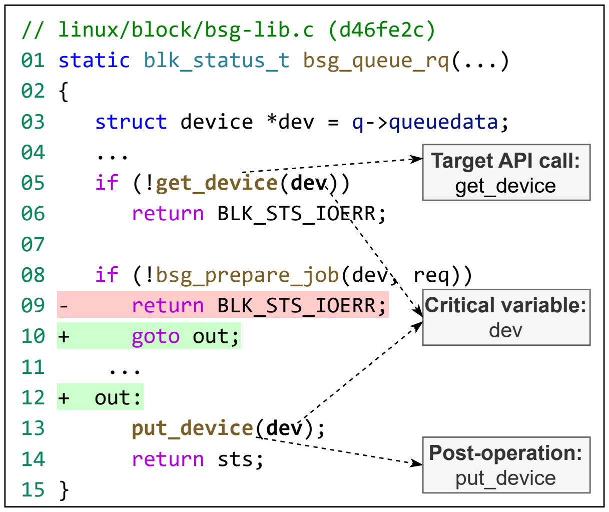
Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242001
www.ndss-symposium.org

## I. INTRODUCTION

Modern software development offers a wide range of APIs, and developers should adhere to security specifications when using them. Violating these specifications can lead to API misuse, threatening the system's security [8], [15], [26]. Due to the diversity of APIs, their usage specifications vary significantly. One common type of specification is API post-handling [17]. Specifically, after calling certain APIs, developers must perform corresponding post-operations, such as checking return values or releasing resources. API post-handling specifications can be represented as a triplet: <*target API, critical variable, post-operation*>. This triplet indicates that after calling the target API, the specified post-operation should be used to handle the effects on the critical variable.

Improper API post-handling (e.g., error checks and causal functions calls) accounts for 66% of API misuse [8], leading to various bugs (e.g., memory corruption, reference count imbalance, and error handling issues) and severe security impacts (e.g., Use-After-Free, system crash) [6], [25], [31], [38]. For example, Figure 1 shows a patch for an API misuse involving `get_device`, which caused a reference count leak [1]. Before the fix, the function called `get_device` at line 5, but directly returned after catching an error at line 9, without performing the required post-operation `put_device` to release the reference count. The patch fixed this bug by adding the missing `put_device` call. This API misuse leads to reference count imbalance, which may cause security impacts such as memory leaks and Use-After-Free [9], [31].

To detect API misuse, it is necessary to know API specifications, which define correct API usage. Violations of these specifications indicate misuse. For this purpose, various methods propose to extract API usage specifications from artifacts such as documentation, API usage, and patches. Accordingly, these methods can be categorized into documentation-based, usage-based, and patch-based. Documentation-based methods leverage natural language processing to extract specifications from API documentation [24], [30], [42]. Usage-based methods mine frequent patterns from API usage code to infer specifications [4], [12], [23], [13], [2]. Patch-based methods extract specifications from historical patches [17].



```
// linux/block/bsg-lib.c (d46fe2c)
static blk_status_t bsg_queue_rq(...)
{
   struct device *dev = q->queuedata;
   ...
   if (!get_device(dev))
      return BLK_STS_IOERR;

   if (!bsg_prepare_job(dev, req))
-     return BLK_STS_IOERR;
+     goto out;
    ...
+  out:
      put_device(dev);
      return sts;
}
```

Fig. 1: A patch fixing misuse of get_device in the Linux kernel

Although these methods successfully extract specifications from corresponding API artifacts, they are inherently limited by the quality and availability of these artifacts. Specifically, documentation-based methods fail when documentation is incomplete or missing. Usage-based methods struggle when API calls are too rare to identify frequent patterns. Similarly, patch-based methods can only extract specifications with relevant patches. As a result, these methods only cover APIs with high-quality artifacts, missing many specifications and leading to false negatives in bug detection.

To address this gap, the goal of this paper is to generate API specifications even in the absence of high-quality API artifacts. Specifically, modern software systems often adopt a hierarchical API design, where high-level APIs build on low-level ones to achieve complex functionality. As a result, high- and low-level APIs tend to share similar specifications. Building on this observation, we propose the idea of *API Specification Propagation*, where specifications propagate through API call chains. For instance, when a high-level API wraps a low-level API, it may inherit the low-level API's specifications. Leveraging this idea, we can generate specifications for APIs that are missed by artifact-based methods, using the known specifications of related high- or low-level APIs.

To achieve this, we propose APISpecGen, a framework designed to iteratively generate API specifications by analyzing propagation relationships. Starting with seed API specifications, APISpecGen performs bidirectional propagation analysis for specification generation and uses newly inferred specifications as seeds for iteratively analysis. APISpecGen consists of two main steps: Specification Propagation Analysis and Specification Generation. In Specification Propagation Analysis, APISpecGen analyzes how API specifications propagate through call chains. Given a set of seed APIs and critical variables, it identifies which APIs the specifications may propagate to (successors) or originate from (predecessors). This involves both caller and callee analysis, resulting in a set of inferred APIs and their critical variables. In the Specification Generation, for each inferred API, APISpecGen determines the corresponding post-operation by analyzing its usage and validating data flows. Using the post-operations from the seed specifications, APISpecGen identifies and confirms the correct post-operations to generate new specifications. To broaden API coverage, APISpecGen employs a bidirectional iterative process, where newly generated specifications serve as seeds for further propagation and generation. This process continues until no new specifications are generated or the iteration limit is reached. Finally, APISpecGen applies these generated specifications to detect API misuse.

We evaluated APISpecGen on the large-scale and widely-used program, the Linux kernel. Starting with 6 seed specifications, APISpecGen generated 7332 new specifications. Leveraging these specifications, APISpecGen identified 186 previously unknown bugs, including reference count leaks, memory leaks, and null pointer dereferences. Among these, 113 bugs have been confirmed by developers, with 8 CVEs assigned. To compare its performance, we also evaluated existing specification extraction methods, including Advance [24], APHP [17], and SinkFinder [5]. These methods, which rely on various API artifacts, covered at most 21% of the specifications generated by APISpecGen. In addition, we used the generated specifications to assess the quality of API artifacts, such as documentation and usage patterns. We found that 66% of API documents lack descriptions about specifications, and we identified 3 errors in the API document. Furthermore, 94% of the generated specifications did not meet the occurrence thresholds required by existing usage-based mining methods. These findings highlight the limitations of artifact-based methods in extracting comprehensive specifications and demonstrate the effectiveness of APISpecGen.

**Contributions.** We summarize our contributions below.

- **New Ideas**: We introduce *API Specification Propagation*, a idea where API specifications propagate through hierarchical API call chains. To formalize this, we present the propagation model and define the conditions for propagation.
- **New Techniques**: We design APISpecGen, a framework that generates new API specifications from seed specifications. By incorporating bidirectional iterative propagation analysis, APISpecGen effectively generates accurate specifications for a wide range of APIs.
- **New Discoveries**: Given 6 seed specifications, APISpecGen generated 7,332 new API specifications. Most of these are beyond the scope of state-of-the-art methods due to limitations in API artifacts. Using these specifications, APISpecGen detected 186 new bugs in the Linux kernel, with 8 assigned CVEs.

## II. Background and Motivation

### A. API Post-handling and Specification

As software systems grow increasingly complex, developers design a wide variety of APIs to support diverse functionalities. At the same time, to ensure system security and stability, API users should adhere to usage specifications when using certain APIs. One critical aspect of this is API post-handling, which involves managing the side effects of certain API calls.

These side effects often require developers to perform specific operations, such as checking return values or pairing resource requests with corresponding release functions.

Specifically, return value checks are essential because many APIs return error values to indicate failure states. Properly handling these return values ensures that errors are caught and addressed. Improper return value checks can confuse the program's logic, leading to system crashes or unintended behaviors [11]. Similarly, APIs that allocate resources, such as memory, reference counts, or file handles, must be paired with corresponding release functions to avoid resource mismanagement. Failure to release these resources can lead to issues such as memory leaks [20], [25], use-after-free [21], [31], or even deadlocks in the case of improperly managed file access or lock APIs. Therefore, detecting API misuse is critical for maintaining security. To detect such bugs, we first need to obtain the corresponding API specifications. Violations of these specifications indicate the presence of bugs.

Following previous work [17], we represent API post-handling specifications using a triplet: *<target API, critical variable, post-operation>*. This triplet specifies that after calling the target API, the post-operation must be performed on the critical variable. The critical variable is recorded based on its position in the target API call. Here, we omit path conditions. For return value checks, the check occurs immediately after the API call, and requiring no additional path conditions. For paired function calls, in most cases, after the target API is successfully called, a resource is acquired and accessed using the critical variable, which must be properly released after use. For example, in Figure 1, the patch bug implies the specification `<get_device, arg1, put_device>`. This means that after calling the `get_device` API, the `put_device` function should be called on its first argument to properly release the reference count once it is no longer needed.

## B. Existing Work and Limitations

To detect API misuse, existing studies propose to extract API specifications from various artifacts, including API documentation [24], [30], [42], API usage code [12], [20], [23], [29], [5], and historical bug patches [17]. Specifically, documentation-based methods, such as Advance [24], use natural language processing techniques to extract specifications from API documentation. API usage-based methods, like APP-Miner [12], rely on frequent pattern mining. Patch-based methods, like APHP [17], extract specifications from bug patches. Although these methods can successfully extract specifications from their corresponding API artifacts, they are inherently limited by the quality and availability of those artifacts. If the artifacts for a given API are incomplete or inadequate, the specification cannot be extracted. For example, documentation-based methods are ineffective when documentation is incomplete or inaccurate. API usage-based methods depend on APIs being used frequently and correctly in the code, so these methods fail if an API is rarely used or often misused. Similarly, patch-based methods rely on the availability of relevant historical patches, and without them, specifications cannot be extracted. In summary, the effectiveness of current methods for extracting API specifications is limited by the quality of the available artifacts. As a result, many API specifications remain uncovered, causing false negatives in bug detection.

## C. Hierarchy in API Design

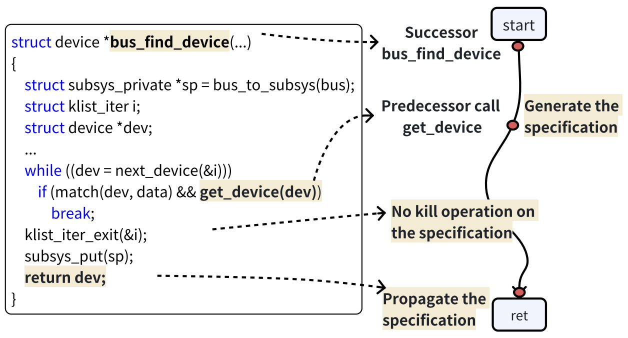
Different APIs in programs are often closely related. Specifically, in software development, API designers often design various APIs for different functions and scenarios. These APIs are typically hierarchical, with upper-level APIs built upon lower-level ones. This hierarchical design allows APIs to serve a wide range of functionalities, each suited to specific use cases. Beyond their functionalities, this hierarchy also establishes relationships between API specifications. Specifically, when upper-level APIs are built on lower-level APIs, they may share similar usage specifications. For example, in glibc [33], the API `g_new` requires calling `g_free` to release the allocated memory. Similarly, `g_strdup` also requires calling `g_free` to free the memory it allocates. In fact, `g_strdup` is wrapped around `g_new` for string duplication, and it inherits the specification from `g_new`. Inspired by the hierarchical nature of API design, we aim to leverage known API specifications as seeds to infer specifications for other APIs.

## III. API Specification Propagation

We observe that API specifications can propagate between APIs through API call chains, a process we refer to as *API Specification Propagation*. During this process, we define the two involved APIs as the *predecessor* and the *successor*, where the successor inherits specifications from the predecessor. This propagation is linked to the flow of critical variables through the API call chain. For example, Figure 2 shows an instance of specification propagation: `get_device` is the predecessor, and `bus_find_device` is the successor, with the critical variable being `dev`. Specifically, in the `bus_find_device`, the call to `get_device` affects the critical variable and subsequently needs to call `put_device`. The critical variable `dev` is returned as a return value. This return not only propagates the variable but also its associated specification. Thus, the specification propagates from `<get_device, arg1, put_device>` to `<bus_find_device, retval, put_device>`. Consequently, after calling `class_put_device`, the same `put_device` operation must be performed on its return value to release the reference count.

To determine if specifications propagate between two APIs, we first define the conditions required for specification propagation. While the successor must call the predecessor, not all callers inherit its specifications. Specifically, specification propagation should meet the following three conditions.

① **Successor calls the predecessor**: The successor must call the predecessor. This call relationship forms the basis for spec-



```
struct device *bus_find_device(...)
{
  struct subsys_private *sp = bus_to_subsys(bus);
  struct klist_iter i;
  struct device *dev;
  ...
  while ((dev = next_device(&i)))
    if (match(dev, data) && get_device(dev))
      break;
  klist_iter_exit(&i);
  subsys_put(sp);
  return dev;
}
```

start
Successor
bus_find_device
Predecessor call
get_device
Generate the specification
No kill operation on the specification
Propagate the specification
ret

Fig. 2: Example of API specification propagation

ification propagation. For example, in Figure 2, the successor `bus_find_device` calls the predecessor `get_device`.

② **Propagation of the critical variable**: Specification propagation relies on variable propagation. The critical variable from the predecessor must be passed to the successor, either as an argument or a return value. For example, in Figure 2, the critical variable `dev` in `get_device` call is also the return value of `bus_find_device`, satisfying this condition.

③ **No operations affecting the critical variable**: After the predecessor's call, no operations affect the critical variable, as these would interrupt specification propagation. We refer to such operations as kill operations. Kill operations have two main types: assignment operations and API post-operations. Specifically, assignment operations reassign the critical variable, nullifying the predecessor API's effects and its specification. API post-operation handles the critical variable according to the specification, preventing further propagation. For example, in Figure 2, after `bus_find_device` calls `get_device`, there are no further operations on `dev`, allowing the specification to propagate.

With API specification propagation, we can use the given seed specifications to perform propagation analysis and identify related predecessors or successors. This allows us to obtain specifications for APIs not covered by existing studies due to the limitations of API artifacts. The idea is shown in Figure 3.

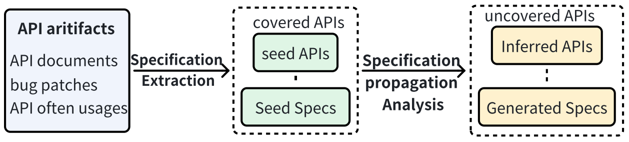


Fig. 3: The application of API specification propagation

## IV. Overview

Based on the above idea, we propose APISpecGen, which performs specification propagate analysis to generate new specifications from seed specifications. The workflow is shown in Figure 4. APISpecGen includes two main phases: *Specification Propagate Analysis* and *Specification Generation.*

In the *Specification Propagate Analysis* phase, APISpecGen analyzes which APIs the seed API specification might propagate to or from, obtaining a list of inferred APIs along with their associated critical variables. This analysis involves both caller and callee specification propagation analysis. In caller analysis, APISpecGen examines the callers of the seed API to determine if they meet the conditions for specification propagation, identifying potential successor APIs, i.e., which APIs the seed specifications might propagate to. In callee analysis, APISpecGen inspects the seed API's source code to find API calls that influence critical variables, identifying potential predecessor APIs. After this, APISpecGen obtains the potential predecessors and successors for the given seed specification, which are referred to as inferred APIs. APISpecGen also records the relative positions of their critical variables for subsequent specification generation.

After identifying a set of inferred APIs that may have associated specifications, APISpecGen determines the appropriate post-operations for these APIs, as they may differ from the seed post-operations. In particular, APISpecGen utilizes actual API usage and data-flow validation to identify the correct post-operations. Specifically, given an inferred API, the critical variable, and the seed post-operation, APISpecGen checks for code that adheres to the inferred specification. If such code is found, the seed post-operation is confirmed as the correct post-operation for the inferred API. If no exact match is found, APISpecGen analyzes the usage of the inferred API to identify candidate post-operations, it then checks the relationship between these candidate post-operations and the seed post-operation to identify the correct post-operation. Finally, APISpecGen generates specifications for the inferred APIs, which include three elements: the inferred API, the post-operation, and the critical variable. If no corresponding post-operation is identified, APISpecGen does not generate a specification for that API. APISpecGen uses the generated specifications as seed specifications to generate new specifications for additional APIs iteratively. This continues until no new specifications are generated or a preset iteration limit is reached. Finally, APISpecGen utilizes the generated API specifications for bug detection.

Figure 5 illustrates a working example of APISpecGen. Starting with the seed specification `<get_device, arg1, put_device>`, APISpecGen generates specifications for `class_find_device` and `nfc_get_device` through iterative specification propagation analysis. First, as shown in Figure 5 (a), APISpecGen analyzes the caller `class_find_device`. After calling `get_device` at line 10, the code operates on the critical variable `dev` and returns it. Since no subsequent operations affecting `dev`, specification propagation occurs, with `dev` passed as the return value in line 15. APISpecGen then verifies that the actual API usage adheres to the specification `<class_find_device, retval, put_device>` and generates the corresponding specification. Next, APISpecGen uses this generated specification as a seed for further propagation analysis, identifying that it also propagates to `nfc_get_device`, where `dev` is again the return value. Further analysis determines that the post-operation for this API is `nfc_put_device`, and the generated specifications are shown in Figure 5 (b).

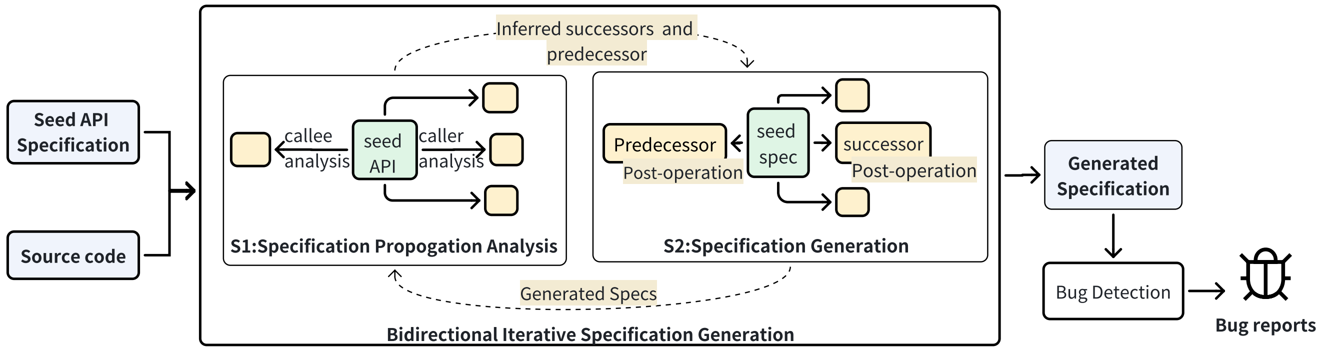


Fig. 4: Overview of APISpecGen

After this, APISpecGen uses the generated specifications for bug detection. Figure 5 (c) shows the new bug detected using the undocumented specification. This bug occurs in the `nfc_genl_vendor_cmd` function within the `net/nfc/netlink.c` file of the Linux kernel. Line 06 calls `nfc_get_device` increments the reference count of the critical variable `dev`. However, after usage (at lines 08, 14, and 17), the reference count is not properly decremented before returning, leading to a reference count leak. APISpecGen detected the bug using the generated specifications. We submitted a patch to fix this, which have been accepted.

## V. Design

Given a seed API specification, consisting of the seed API, critical variable, and post-operation, APISpecGen conducts analysis in two stages. It first performs specification propagation analysis to uncover hidden APIs and then infer their corresponding post-operations.

### *A. Specification Propagation Analysis*

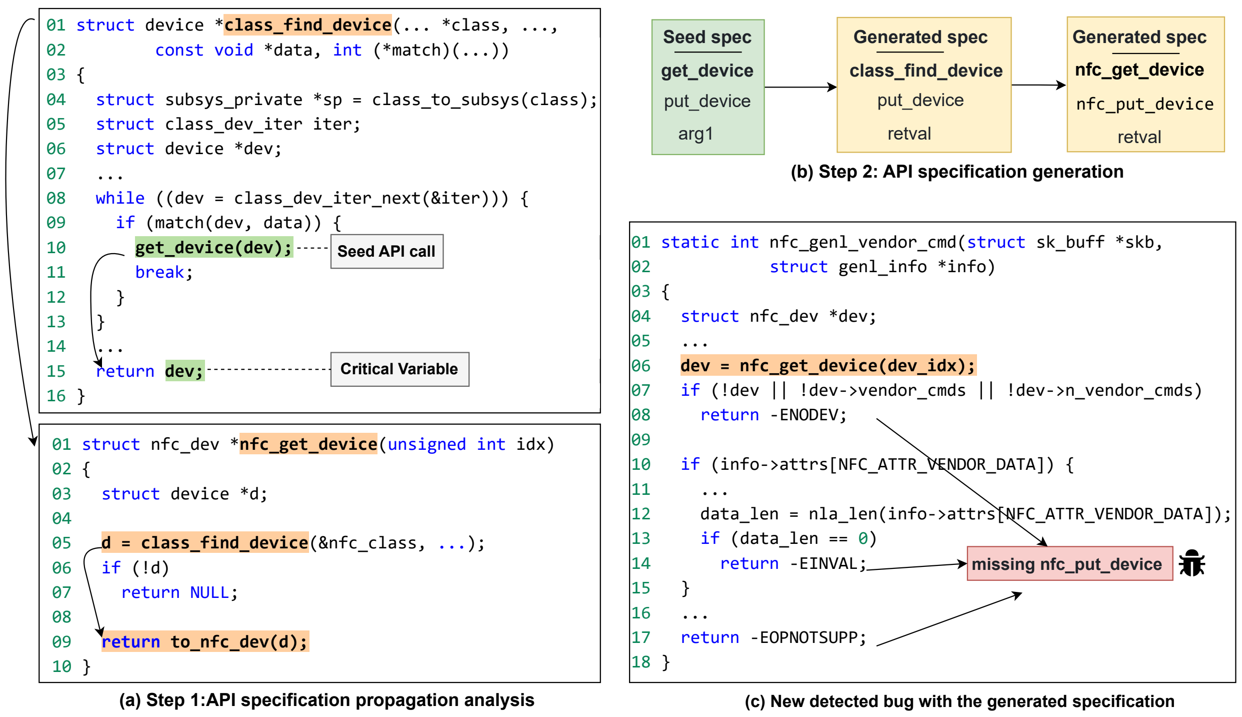
To analyze specification propagation, APISpecGen identifies both the APIs from which the seed specification may originate (predecessors) and the APIs to which it may propagate (successors). A straightforward approach to analyzing the propagation of a seed API's specification is identifying API wrappers—functions that simplify API calls and often inherit their specifications. Typically, this is done through parameter matching, where the wrapper's parameter list is compared to the original API's. However, this method is both inefficient and prone to false positives and false negatives. Firstly, many APIs that inherit a seed API's specification are not simple wrappers. They extend the seed API for different scenarios, often with parameter lists that do not match the seed API's. Analyzing only API wrappers would lead to many missed specifications. On the other hand, not all wrappers inherit the specifications. Some are specifically designed to handle API specifications internally for easier use and do not propagate specifications. Therefore, relying solely on wrapper detection can cause both false negatives and false positives for specification generations.

To generate new specifications using the seed specification, APISpecGen conducts a bidirectional specification propagation analysis. As mentioned earlier, the propagation of API specifications occurs from one API as a predecessor to another API as a successor. A given API can act as both a predecessor and a successor in the specification propagation chain. Therefore, APISpecGen traces specification propagation in both directions: identifying successors by analyzing the callers of the seed API and finding predecessors by analyzing its callees. Next, we introduce these two analysis separately.

**Caller Analysis to Get Successors.** APISpecGen analyzes the callers of the seed API to determine where the seed specification propagates, i.e., its successors. Specifically, APISpecGen examines each caller to verify if it satisfies the conditions for specification propagation. The process begins by identifying all the callers of the seed API. For each caller, APISpecGen further checks if the conditions ② and ③ are met for specification propagation.

To check the condition ②, which requires the critical variable to propagate to the caller as either an argument or a return value. APISpecGen locates the seed API call and identifies the critical variable name based on its relative position to the seed API. Using abstract syntax tree (AST) parsing, APISpecGen retrieves the variable's name in the caller's code and verifies whether it appears as a parameter or return value. If not, the specification cannot propagate to this caller. The lightweight name-based approach enables efficient analysis for large-scale programs because developers typically use distinct names for different variables within functions. To evaluate its accuracy, we randomly selected 100 callers of a seed API and manually analyzed whether the specifications propagated correctly. The results showed only 7% false negatives and no false positives for determining specification propagation.

Once the condition ② is satisfied, APISpecGen checks the condition ③, ensuring that no further operations affect the critical variable after the seed API call. To do this, it analyzes the caller's Control Flow Graph (CFG) and performs topological sorting to ensure the correct order of operations. It then inspects nodes following the seed API call to detect any reassignments or post-operations on the critical variable. Such operations would terminate the specification propagation.


```
struct device *class_find_device(... *class, ...,
        const void *data, int (*match)(...))
{
  struct subsys_private *sp = class_to_subsys(class);
  struct class_dev_iter iter;
  struct device *dev;
  ...
  while ((dev = class_dev_iter_next(&iter))) {
    if (match(dev, data)) {
      get_device(dev);
      break;
    }
  }
  ...
  return dev;
}
```

```
struct nfc_dev *nfc_get_device(unsigned int idx)
{
  struct device *d;

  d = class_find_device(&nfc_class, ...);
  if (!d)
    return NULL;

  return to_nfc_dev(d);
}
```

(a) Step 1:API specification propagation analysis

(b) Step 2: API specification generation

```
static int nfc_genl_vendor_cmd(struct sk_buff *skb,
          struct genl_info *info)
{
  struct nfc_dev *dev;
  ...
  dev = nfc_get_device(dev_idx);
  if (!dev || !dev->vendor_cmds || !dev->n_vendor_cmds)
    return -ENODEV;

  if (info->attrs[NFC_ATTR_VENDOR_DATA]) {
    ...
    data_len = nla_len(info->attrs[NFC_ATTR_VENDOR_DATA]);
    if (data_len == 0)
      return -EINVAL;
  }
  ...
  return -EOPNOTSUPP;
}
```

(c) New detected bug with the generated specification

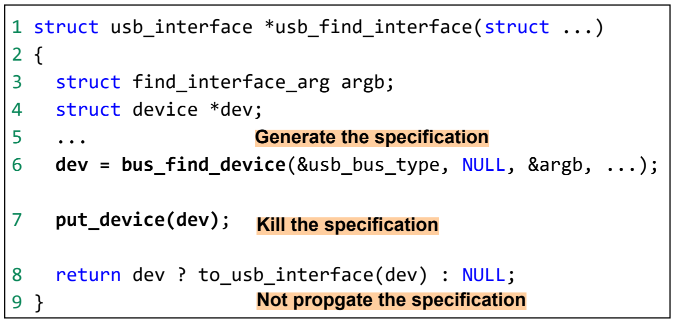
Fig. 5: The working example of APISpecGen for specification generation and bug detection

In some functions, post-operations may only appear in some paths. To address this, APISpecGen distinguishes between normal and error paths in callers, as resource post-handling is usually needed when API calls succeed. APISpecGen checks if post-operations (e.g., put_device) are missing from all normal paths for propagation. Inconsistent post-handling between normal paths usually indicates bugs [20], not specification propagation. To evaluate the accuracy of this path-sensitive analysis strategy, we conducted a pre-study by randomly selecting 100 callers of seed APIs. Our study shows 100% accuracy of this design in identifying successors. Additionally, in some cases, the program may perform multiple operations on different members of the same structure—such as allocating resources to each member, where each operation requires its own post-operation. APISpecGen performs field-sensitive analysis for callers and can differentiate operations involving different variables from the same structure.

Callers meeting both conditions are recorded as potential successors, along with the positions of their critical variables. In cases where functions contain multiple target APIs or post-operations, these typically involve different critical variables or follow distinct paths. APISpecGen analyzes each critical variable and then checks each path separately, thus differentiating these operations without impacting results.

For example, as shown in Figure 5, the function class_find_device calls get_device. APISpecGen inspects the source code of class_find_device to check if it inherits the specifications of get_device. Since condition ① is already met (the successor calls the predecessor), APISpecGen proceeds to check conditions ② and ③. To satisfy the condition ②, the critical variable must propagate to the caller as its arguments or return value. APISpecGen locates the call to the seed API get_device and identifies the critical variable dev using its relative position (arg1). Because dev is also returned by class_find_device, condition ② is met. Next, APISpecGen checks condition ③ by analyzing the CFG of class_find_device. The analysis confirms that after calling get_device, no further operations affect dev, meaning condition ③ is also satisfied. Thus, APISpecGen determines that class_find_device inherits the specification from get_device. However, in some cases, even when the critical variable propagates, the specification may not. For instance, as shown in the Figure 6, bus_find_class inherits the specification of get_device, and usb_find_interface is the caller of bus_find_class. Here, the critical variable dev is returned, satisfying condition ②. Further analysis, however, shows that after calling bus_find_device, the caller performs the post-operation put_device on dev at line 7, condition ③ is not met. In this case, although the variable propagates, the specification does not propagate.

**Callee Analysis to Get Predecessor.** To identify the predecessor for a given specification, APISpecGen analyzes the seed API's callees related to the critical variable. Typically, a specification is propagated by a single API, so the goal is to find a unique callee as the predecessor. To do this,



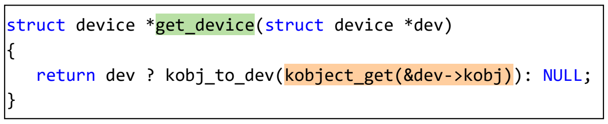
```
struct usb_interface *usb_find_interface(struct ...)
{
  struct find_interface_arg argb;
  struct device *dev;
  ...
  dev = bus_find_device(&usb_bus_type, NULL, &argb, ...);

  put_device(dev);

  return dev ? to_usb_interface(dev) : NULL;
}
```

Fig. 6: Caller analysis for bus_find_device

APISpecGen conducts callee analysis by tracing upward from the function's return point in the seed API's source code. During this process, APISpecGen employs a field-insensitive analysis for structures. APISpecGen first looks for the last assignment to the critical variable related to a function call. If such an assignment is found, the associated function is considered the predecessor. If no direct assignment is found, APISpecGen examines functions that pass the critical variable as an argument. Any function accepting the critical variable as an argument could potentially modify it. If only one such function exists, it is considered the predecessor.

To reduce noise during analysis, we pre-collected a set of commonly used macros (e.g., `kobj_to_dev`) that do not affect variables. They can be safely ignored when identifying predecessors. If a unique predecessor cannot be determined (e.g., multiple operations may affect the same variable), APISpecGen focuses solely on successors, as seed specifications may stem from multiple operations. To evaluate this design, we conducted a pre-study with 100 randomly selected target APIs. The results show that APISpecGen successfully identifies 91% of predecessors with no false positive.

For example, as shown in Figure 7, APISpecGen analyzes the source code of `get_device` and finds that no function directly assigns the critical variable. Instead, it identifies `kobject_get`, which takes the critical variable as an argument. During the analysis, APISpecGen recognizes `kobj_to_dev` as a pre-collected macro which does not affect the variable. Based on this, APISpecGen determines that `kobject_get` is the predecessor for the `get_device` specification and records `arg1` as the critical variable.



```
struct device *get_device(struct device *dev)
{
    return dev ? kobj_to_dev(kobject_get(&dev->kobj)): NULL;
}
```

Fig. 7: Callee analysis to get the predecessor of get_device

## B. Specification Generation

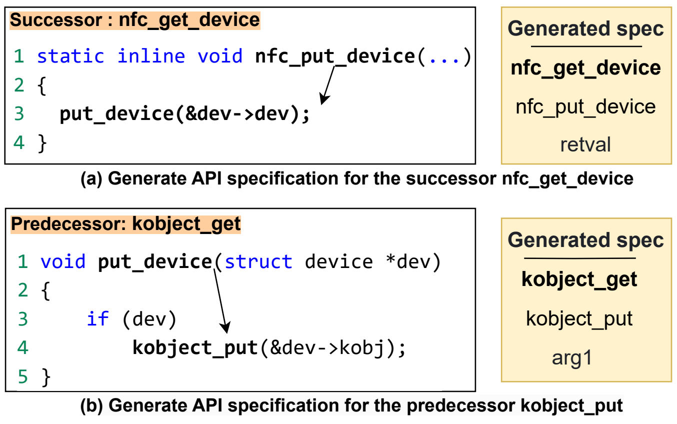
In the previous step, APISpecGen identifies a set of inferred APIs and their corresponding critical variables. Next, APISpecGen determines the post-operations for these APIs, as they may differ from the seed post-operations. Similar to the caller-callee relationships between seed and inferred APIs, the post-operations also follow similar patterns and may propagate the critical variable.

To determine the correct post-operation, APISpecGen combines the usage of the inferred APIs with data-flow validation. Specifically, for each inferred API, its critical variable, and the seed post-operation, APISpecGen first checks if the seed post-operation is applicable by querying the API usage in the program. If the usage matches the specification, the seed post-operation is validated as the correct post-operation for the inferred API. APISpecGen performs a deeper analysis to identify the correct post-operation for the inferred API. It examines the API usage to collect candidate post-operations and validates them through data-flow analysis with the seed to determine the correct one.

To identify candidate post-operations for an inferred API, APISpecGen first examines the callers of the API and locates the points where the inferred API is invoked. It then collects the operations that affect the critical variables after these invocations, which are considered as candidate post-operations. Next, APISpecGen analyzes the relationship between these candidate post-operations and the seed post-operation to identify the correct post-operation. The two should follow similar caller-callee relationships and propagate the critical variable. Specifically, if the inferred API calls the seed API, its corresponding post-operation should also call the seed post-operation. APISpecGen then checks inside the candidate post-operation to see if it propagates the critical variable to the seed post-operation. If it does, the operation is confirmed as the post-operation for the inferred API. Similarly, if the inferred API is called by the seed API, APISpecGen inspects the seed post-operation to check if it propagates the critical variable to any candidate post-operation. If so, that operation is determined as the post-operation for the inferred API. If no corresponding post-operation is found, APISpecGen does not generate a specification for the inferred API. By combining API usage analysis with data-flow validation, APISpecGen ensures the accuracy of the generated specifications.

As shown in Figure 5, during the specification propagation analysis, APISpecGen identifies `class_find_device` as the successor of `get_device`, with `put_device` as the seed post-operation. By analyzing its usage, APISpecGen confirms that the post-operation of `class_find_device` is indeed `put_device`, thus generating the corresponding specification. Next, APISpecGen uses this post-operation to generate the specification for `nfc_get_device`. It begins by analyzing the usage of `nfc_get_device` and identifying candidate post-operations, including `nfc_put_device`. Since `nfc_get_device` calls `class_find_device`, APISpecGen checks whether the relationship between `nfc_put_device` and `put_device` mirrors the caller-callee relationship of the critical variable transfer. Upon finding that `nfc_put_device` passes the critical variable `dev` to `put_device`, APISpecGen infers that `nfc_put_device` is the correct post-operation for `nfc_get_device`, thereby generating its specification.

Similarly, in the specification generation for the predecessor, APISpecGen first identifies the API `kobject_get` and its critical variable `arg1`. During the usage analysis of `kobject_get`, APISpecGen identifies candidate post-operations, including `kobject_put`. Since `kobject_get` is called by `get_device`, APISpecGen further analyzes `kobject_put` and observes that the critical variable `dev` is passed to it. This establishes a connection between the candidate post-operation `kobject_put` and the seed post-operation `put_device`. Based on this, APISpecGen infers `kobject_put` as the post-operation for `kobject_get` and generates the corresponding specification.



**Successor : nfc_get_device**

```
static inline void nfc_put_device(...)
{
  put_device(&dev->dev);
}
```

**Generated spec**

**nfc_get_device**

nfc_put_device

retval

**(a) Generate API specification for the successor nfc_get_device**

**Predecessor: kobject_get**

```
void put_device(struct device *dev)
{
    if (dev)
        kobject_put(&dev->kobj);
}
```

**Generated spec**

**kobject_get**

kobject_put

arg1

**(b) Generate API specification for the predecessor kobject_put**

Fig. 8: Specification generation for the inferred APIs

## C. Iterative Specification Propagation Analysis

Through specification generation, APISpecGen generates specifications for APIs that are either callers or callees of the seed API. To maximize the coverage of APIs and effectively utilize the seed specifications, APISpecGen employs a iterative propagation analysis. It treats each newly generated specification as a seed for further analysis, thus can explore a broader range of APIs. Specifically, in the caller propagation analysis, APISpecGen investigates not only the direct callers of the seed API but also the indirect callers along the call chain. On the other hand, in callee analysis, APISpecGen digs deeper into lower-level APIs, these APIs are then used to generate specifications for higher-level APIs that may have been missed initially. By iterating through this process, APISpecGen can uncover new predecessors and successors, extending beyond the callers and callees of the initial seed API.

For example, Figure 9 illustrates the API specification propagation graph generated by iteratively using `get_device` as the initial seed. Through callee analysis of `get_device`, APISpecGen generates the specification for the lower-level API `kobject_get`. This specification is then used as a new seed for further propagation. As a result, APISpecGen identifies that the `kobject_get` specification also propagates to `of_node_get`. Further iterations lead to the generated specification for `of_irq_find_parent` as shown in Figure 9. Figure 10 illustrates a bug detected using these generated specifications. Specifically, in the `mvebu_gicp_probe` function, after calling `of_irq_find_parent` on line 06, a node with an incremented reference count is returned. According to the generated specification, the function should call `of_node_put` to release the reference count of the critical variable `irq_parent_dn` once the node is no longer needed. However, the function does not release the reference, leading to a reference count leak. Although `of_irq_find_parent` has no caller-callee relationship with the `get_device` API, APISpecGen can generate the specification for it through iterative propagation. This specification is also undocumented. As a result, APISpecGen successfully identified the bug that existing tools had missed. We have submitted a patch to fix this bug, which has been accepted by the developers.

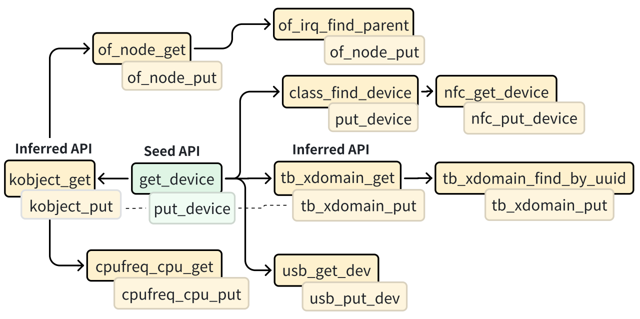
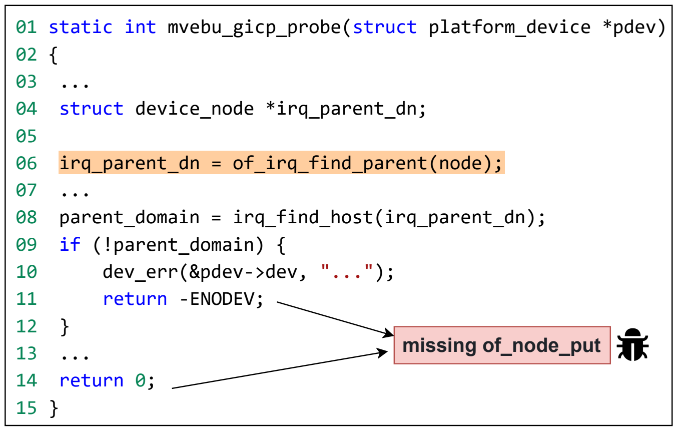


Fig. 9: API specification propagation graph of get_device



```
01 static int mvebu_gicp_probe(struct platform_device *pdev)
02 {
03  ...
04  struct device_node *irq_parent_dn;
05
06  irq_parent_dn = of_irq_find_parent(node);
07  ...
08  parent_domain = irq_find_host(irq_parent_dn);
09  if (!parent_domain) {
10      dev_err(&pdev->dev, "...");
11      return -ENODEV;
12  }
13  ...
14  return 0;
15 }
```

Fig. 10: Detected misuse of the of_irq_find_parent API

## VI. Implementation

We implemented APISpecGen in Python. It includes two main modules: specification generation and bug detection. For code analysis, we utilize three tools: Joern [39], Weggli [37], and tree-sitter [34]. Specifically, Joern generates Program Dependence Graphs, enabling analysis of data flow and control flow within functions. Weggli allows rapid code querying based on Abstract Syntax Trees (ASTs), and tree-sitter parses

ASTs to identify specific statements or variables. Additionally, we've tried LLVM for static analysis. However, it's difficult to cover architecture-specific and hardware-dependent code in operating system such as the Linux kernel. Even with `allyesconfig`, it covers only 60%–70% of the code [32]. In contrast, our implementation does not require compiling the entire codebase, allowing us to analyze all source files comprehensively. To simplify loop handling, we replace loops with if statements and unroll them once.

For bug detection, we integrate APHP's bug detector by configuring path conditions for generated specifications. Specifically, return value checks are performed immediately after the API returns, thus no need for additional path conditions. For paired function calls, we assume that the API calls succeed. In such cases, once the target API successfully allocates resources, post-operations are applied to the critical variable when it is no longer needed. To reduce false positives, we perform variable escape analysis: if variables are returned via return values or parameters, they are not reported as bugs.

## VII. Evaluation

In experiments, we first evaluated the effectiveness of APISpecGen in generating specifications and its ability to detect unknown bugs. Additionally, we evaluated how well existing methods cover the specifications and bugs identified by APISpecGen. Based on the generated specifications, we further analyzed the quality of API artifacts, including API documentation, API usage, and API names, highlighting their limitations when used for extracting API specifications.

**Dataset and Seeds.** We evaluated APISpecGen on the Linux kernel (version v5.16), a widely-used open-source project with over 20 million lines of code and diverse APIs. This large-scale and complex program provides an ideal benchmark for testing APISpecGen. Besides, APISpecGen works without requiring pre-compilation or additional configuration.

APISpecGen is fully automated and uses APHP to extract seed specifications from bug patches. Each specification is represented as a triplet, including the target API, critical variable, and post-operation. For evaluations, we collected six specifications from Linux kernel patches, as shown in Table I. These specifications represent different API types, including five related to paired function calls and one related to return value checks. Specifically, APIs such as `get_device`, `device_initialize`, and `try_module_get` handle reference counting, while `kmalloc` and `kstrdup` focus on memory allocation. The remaining specification, `ERR_PTR`, is related to return value checks. This API converts error numbers into error pointers, requiring the corresponding post-operation `IS_ERR` to validate the return value.

**Platform.** The experiments were conducted on a 64-bit Ubuntu 22.04 system with 503GB of memory and 5TB of storage space, powered by an Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz with 79 processors.

### A. Effectiveness of Specification Generation

For specification generation, we considered all functions in the program as APIs and limited the iterative specification generation process to a maximum of 10 iterations. APISpecGen is fully automated, and does not require manual effort in the process. Using the six seed specifications, APISpecGen successfully generated 7332 new specifications, as summarized in Table II. We manually check whether the specifications are correct by examining the function names, semantics and usage through the available code and documentation. As verifying each specification is time-consuming, we randomly selected 200 specifications for each seed that generated more than 200 specifications. In total, we spent about 13 person-hours verifying 763 sampled specifications, all of which were correct, averaging about one minute per check. We believe that this is manageable as a one-time effort. In particular, most function names provide valuable semantic clues for verification—about 87% of function pairs have informative keywords (e.g., get/put) and follow similar structures. For uncertain cases, we analyze operations on critical variables to verify the propagation. This typically involves only a few lines of code and requires minimal effort.

TABLE I: Seed specifications and their original commits

| Target API | Post-Operation | Var | Commit |
|---|---|---|---|
| get_device | put_device | arg1 | d46fe2cb2dc |
| device_initialize | put_device | arg1 | a5808add9a6 |
| try_module_get | module_put | arg1 | 44f8baaf230 |
| kmalloc | kfree | retval | 493ff661d43 |
| kstrdup | kfree | retval | e629e7b525a |
| PTR_ERR | IS_ERR | arg1 | 59715cffce1 |

TABLE II: Results of generated specifications and new bugs

| Seed API | #Spec | #New Bug | Bug Type |
|---|---|---|---|
| get_device | 760 | 137 | refcount leak |
| device_initialize | 91 | 6 | refcount leak |
| try_module_get | 58 | 1 | refcount leak |
| kmalloc | 1202 | 30 | memory leak |
| kstrdup | 14 | 1 | memory leak |
| ERR_PTR | 5207 | 11 | NULL-pointer-deref |
| **Total** | **7332** | **186** | - |

The evaluation results confirm the correctness of the generated specifications. Although propagation analysis might introduce inaccuracies due to imprecise data flow analysis, the specification generation step filters out these inaccuracies. In this step, APISpecGen correlates the actual API usage and the data flow between generated post-operations and seed post-operations to ensure that specifications are only generated for APIs with successful post-operation identification. We manually analyzed the generated specifications and found that they cover a wide range of APIs. Based on these specifications, we have the following two observations.

*1) The inferred API can differ significantly from the original seed API.* For example, Figure 11 illustrates the propagation analysis path starting from the seed API `get_device` to a totally different API, `rdma_user_mmap_entry_get`. Specifically, start with `get_device`, APISpecGen identifies the predecessor API `refcount_inc_not_zero` via iterative callee analysis. Further caller analysis reveals the successor API `rdma_user_mmap_entry_get`, with its post-operation, `rdma_user_mmap_entry_put`, which differs from `put_device`. In this case, APISpecGen successfully generates specifications for `rdma_user_mmap_entry_get` based on the seed API `get_device`, despite the lack of any direct or indirect call relationship between the two APIs. This is possible because APISpecGen focuses solely on code involving the critical variables in the specification, ignoring irrelevant semantics. Its bidirectional iterative generation enables it to trace lower-level APIs from seed APIs and discover related higher-level APIs. In this way, APISpecGen can generate specifications for diverse APIs, regardless of their similarity to the seed.

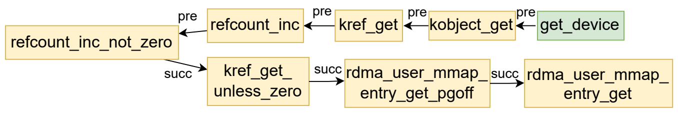
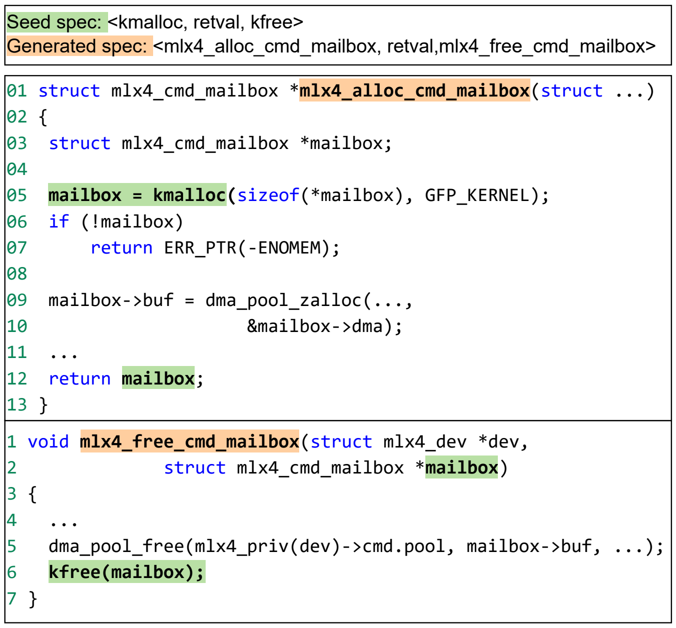


Fig. 11: From get_device to rdma_user_mmap_entry_get

*2) The inferred APIs may inherit and integrate specifications from multiple APIs.* For example, Figure 12 shows the seed specification of `kmalloc` and the generated specification. In the source code of `mlx4_alloc_cmd_mailbox`, this API integrates specifications from both `kmalloc` and `dma_pool_zalloc`. Specifically, it calls `kmalloc` on line 5, assigning the return value to the critical variable `mailbox`, thus inheriting `kmalloc`'s specification. On line 9, it calls `dma_pool_zalloc` to allocate memory for its `buf` field. Thus also inheriting `dma_pool_zalloc`'s specifications. By analyzing the API usage and validating the data flow, APISpecGen identifies `mlx4_free_cmd_mailbox` as the post-operation. This function integrates the post-operations of both lower-level APIs by calling `dma_pool_free` to release the memory allocated by `dma_pool_zalloc` and `kfree` for the memory allocated by `kmalloc`. This example demonstrates APISpecGen's ability to discover integrated specifications that span multiple lower-level APIs.

**Scalability and Generalizability.** APISpecGen demonstrates great scalability in generating API specifications for large-scale programs. Specifically, when applied to the Linux kernel, a codebase with 23 million lines of code, APISpecGen generated 7332 specifications in just 2 hours, averaging only one second per specification, with memory usage remaining under 2 GB throughout the process. APISpecGen focuses on analyzing critical code while ignoring irrelevant parts and uses multithreading to enable parallel processing for efficiency. The results show APISpecGen's ability to handle large-scale programs efficiently.

APISpecGen is applicable to general software, as APIs are typically designed hierarchically for systematic management,


Seed spec: <kmalloc, retval, kfree>
Generated spec: <mlx4_alloc_cmd_mailbox, retval,mlx4_free_cmd_mailbox>

```
01 struct mlx4_cmd_mailbox *mlx4_alloc_cmd_mailbox(struct ...)
02 {
03  struct mlx4_cmd_mailbox *mailbox;
04
05  mailbox = kmalloc(sizeof(*mailbox), GFP_KERNEL);
06  if (!mailbox)
07      return ERR_PTR(-ENOMEM);
08
09  mailbox->buf = dma_pool_zalloc(...,
10                      &mailbox->dma);
11  ...
12  return mailbox;
13 }
```

```
1 void mlx4_free_cmd_mailbox(struct mlx4_dev *dev,
2              struct mlx4_cmd_mailbox *mailbox)
3 {
4   ...
5   dma_pool_free(mlx4_priv(dev)->cmd.pool, mailbox->buf, ...);
6   kfree(mailbox);
7 }
```

Fig. 12: Case: generated specification with kmalloc as seed

allowing API specification propagation. APISpecGen is fully automated using APHP to extract seeds. It can be applied to other software using their patches. To evaluate its generalizability, we applied APISpecGen to OpenSSL [27] and FFmpeg [7], using one seed for each application: `<BIO_new, retval, BIO_free>` for OpenSSL and `<av_malloc, retval, av_free>` for FFmpeg. APISpecGen generated 39 specifications for OpenSSL and 76 for FFmpeg, including specifications such as `<bio_from_file, retval, BIO_free>` in OpenSSL and `<ff_urldecode, retval, av_free>` in FFmpeg. These results show APISpecGen's capability to generate API specifications across diverse software.

## B. Effectiveness of Bug Detection

Using the specifications generated by APISpecGen, we further use them to check violations for bug detection. Through APISpecGen, we identified 186 new bugs. We confirmed the true bugs from 325 bug reports in 6 person hours, with each report taking less than one minute. The verification process is straightforward and manageable. Specifically, the generated bug reports include information about buggy functions and the violated API specifications, and since most API misuses occur within single functions, we can simply compare the functions' API usages with the specifications to confirm violations. The detected bugs correspond to 90 API usage specifications. Table VII shows 60 of these bugs. These bugs are caused by improper API post-handling, such as missing reference count release, missing memory release, and incorrect return value checks. APISpecGen generates specifications for diverse APIs not covered by previous studies, and thus can detect previously undetected bugs.

The security impacts of detected bugs are typically related to the types of API specification violated. Table II categorizes the types of detected bugs based on the specifications generated from different seed APIs. For example, misuse of reference count APIs like `get_device` can result in reference count leaks. Misusing memory allocation APIs such as `kmalloc` may cause memory leaks. Additionally, incorrect handling of return values, such as `PTR_ERR`, can lead to invalid (NULL) pointer dereferences, causing system crashes or instability.

For responsible disclosure, we report detected bugs to developers. Specifically, we carefully analyze the bugs and submit the corresponding patches to assist developers in fixing them. So far, 113 bugs have been confirmed or fixed by developers. We manually analyze the modules where these bugs are located and evaluate their potential impacts. Bugs with severe impacts are prioritized for CVE requests. Table III shows the assigned CVEs, all of which have been fixed in the latest versions with the submitted patches.

TABLE III: The assigned CVEs for found bugs.

| Subsystem | CVE ID | Security Impact | Status |
|---|---|---|---|
| gpu/drm | CVE-2023-22998 | System Crash | Fixed |
| drivers/usb | CVE-2023-22999 | System Crash | Fixed |
| drivers/phy | CVE-2023-23000 | System Crash | Fixed |
| drivers/scsi | CVE-2023-23001 | System Crash | Fixed |
| drivers/bluetooth | CVE-2023-23002 | System Crash | Fixed |
| tools/perf | CVE-2023-23003 | System Crash | Fixed |
| drivers/gpu | CVE-2023-23004 | System Crash | Fixed |
| drivers/gpu | CVE-2023-23006 | System Crash | Fixed |

**False Positives.** Despite the correctness of the specifications, the intra-function analysis in bug detection can still produce false positives. In our evaluation, the false positive rate for bug detection is 42%, which is reasonable for static analysis-based detection in complex programs [20], [17]. There are two main causes for false positives. First, false positives occur when critical variables are post-handled by an operation different from the one specified in the API specifications, even though the operations are essentially equivalent for post-handling. For example, the post-operation may indirectly call the one specified in the specifications. In such cases, intra-function analysis incorrectly reports a missing post-operation. For more accurate detection, the specifications generated by APISpecGen could be integrated with more advanced, inter-procedural bug detectors to reduce false positives. Second, false positives also arise from inaccurate alias analysis. Specifically, if critical variables are aliased by other variables, and the subsequent post-operations are performed on the alias, the incorrect alias analysis leads to false positives.

**False Negatives.** To evaluate false negatives in bug detection, we constructed a bug dataset, following prior work [17]. Specifically, we randomly selected 100 generated specifications and used them to create 100 corresponding bugs. For each specification, we injected bugs by removing the correct post-operation from a randomly chosen target function, causing it to violate the API specifications. The results show that the bug detection recall is 11%. The main cause of false negatives is the escape analysis used to reduce false positives. While escape analysis helps avoid reporting false positives, it can also lead to missed bugs. Specifically, when a variable is propagated as an argument or return value, and it lacks a corresponding post-operation in the current function, the violation is not detected. However, in some cases, even after the variable is propagated, it may still lack the required post-operations, leading to false negatives during detection.

## C. Compared with Related Work

We compared APISpecGen with four methods for API misuse detection, each leveraging different types of artifacts: Advance [24] (documentation-based), IPPO [20] and SinkFinder [3] (API usage-based), and APHP [17] (patch-based). APISpecGen aims to complement these methods by generating specifications for APIs they cannot cover. In this way, we used the bugs detected by APISpecGen and their corresponding specifications as ground truth for comparisons. Specifically, we analyzed how many of these bugs and specifications could be covered by existing methods. If a method could extract a relevant specification, we assumed it could also detect the corresponding bug. The results are shown in Table IV.

TABLE IV: The SOTA's results for the specification and bugs

| **Method** | **Coverage of Spec** | **Coverage of Bugs** |
|---|---|---|
| Advance [24] | 14% (13/90) | 10% (19/186) |
| IPPO [20] | - | 0% (0/186) |
| APHP [17] | 21% (19/90) | 17% (32/186) |
| SinkFinder [3] | 11% (9/(90-11)) | 10% (17/(186-11)) |

Note: IPPO directly employs inconsistency check for bug detection without specification extraction phase.

**Compared with Advance.** Advance [24] extracts specifications from API documentation to detect API misuse. It utilizes sentiment analysis to locate sentences with strong sentiment, which are assumed to describe API usage specifications. To evaluate Advance, we collected all function comments in the Linux kernel, which are also used to generate its official documentation. We checked whether Advance could extract the specification-related sentences from these comments. If Advance identified such sentences, we considered the specification could be covered by Advance. We also analyzed how many bugs could be detected using the specifications extracted by Advance. The results showed that Advance could only extract 14% of the specifications generated by APISpecGen. There are two main reasons for this: *(1) Incomplete documentation*: The API documentation does not include information about post-operations, so Advance misses these specifications. *(2) Strong sentiment requirement*: Advance relies on strong sentiment analysis to locate sentences describing the specifications. However, API documentation may use neutral language to describe API usage, and Advance ignores these sentences.

**Compared with IPPO.** IPPO uses inconsistency checking to detect bugs by comparing operations along different paths within the same function. Although it does not extract API specifications, most of the bugs it detects are due to missed API post-handling, such as missed reference decrements and memory releases. These bug types overlap with those detected by APISpecGen. Therefore, we included IPPO in our comparison. We evaluated how many bugs detected by APISpecGen could also be identified by IPPO. Our results show that IPPO failed to detect any of these bugs, mainly for two reasons: *(1) Lack target APIs Information*: IPPO requires predefined security operations, such as reference count operations and resource operations, which correspond to the target APIs in API specifications. However, identifying these target APIs is challenging because it is not always clear which APIs require post-operations. Without this information, IPPO cannot perform inconsistency checks and fails to detect related API misuses. *(2) Lack comparable path for post-operation*: IPPO requires two comparable paths within a function—one with the post-operation and one without—for inconsistency checks to detect bugs. However, many API misuse scenarios do not meet this condition. For example, the bugs shown in Figure 5 and Figure 10 lack post-operations on all paths, and IPPO cannot detect these bugs through inconsistency checks.

**Compared with SinkFinder.** SinkFinder [3] discovers function pairs based on seed pairs by applying frequent pattern mining to identify suspicious API pairs. It then filters these pairs, retaining only those with semantics similar to the seed pairs. In evaluations, we focus on the specifications and bugs related to function pairs. Based on the seed specifications, APISpecGen generates pairs involving `alloc/free` and `get/put`, which are both considered by SinkFinder. The results show that SinkFinder covers 11% pairs, and detects 10% bugs. SinkFinder fails to generate most function pairs due to two main limitations: *(1) Frequent correct usage requirement*: SinkFinder relies on frequent pattern mining, requiring a function pair to appear at least 10 times in the code. We found that most of the generated specifications do not meet this threshold. Further details are provided in Section VII-D. *(2) Semantic similarity requirement*: SinkFinder generates new pairs using seed pairs, relying on semantic similarity between them. This similarity is calculated based on function names and control flow graph embeddings. However, many pairs generated by APISpecGen have function names and semantics that differ significantly from those of the seed pairs, making them difficult for SinkFinder to identify. In contrast, APISpecGen does not depend on function pair frequency or semantic similarity to seed pairs. As a result, it can detect function pairs that SinkFinder fails to identify.

**Compared with APHP.** APHP extracts API specifications from bug patches. As it is infeasible to collect all patches related to API misuse, we used APHP's patch dataset to evaluate its coverage of APISpecGen's specifications. We also evaluated the false negatives in bug detection caused by APHP's limited specification coverage. The results show that APHP covers only 21% of the specifications generated by APISpecGen, leading to the detection of just 17% of the bugs. APHP can only extract specifications from patches that explicitly fix API misuses. However, bug patches are inherently incomplete, leaving many API specifications uncovered.

### D. The Utilizability of API Artifacts

Existing studies extract API specifications from various artifacts. In particular, function pairs cannot be directly obtained by analyzing a single API's source code. The relationships between function pairs are determined during API design, and analyzing only the target API's source code does not reveal the associated post-operations. To infer these pairs, existing methods use alternative artifacts. API documentation is a direct source of specifications [24]. Frequent usage patterns are considered correct API usage and serve as a basis for extracting specifications [40], [3], [12], [4]. API names can also offer semantics about the API's function, helping identify certain types of operations [25], [35]. To assess the limitations of these methods, we focus on the function pairs generated by APISpecGen and examine the associated artifacts of the corresponding target APIs to determine whether these artifacts can be used to extract the generated pairs.

**The Quality of API Documentation.** We check whether the generated API specifications are mentioned in the API documentation. The results showed that 87% of the specifications were not included in the API documentation. Considering that we treated all functions as APIs, which might include some internal functions not needing documentation, we further analyzed only those APIs with documentation. For these APIs, 66% of the specifications were still not mentioned. The results show the inherent limitation of documentation-based methods due to incomplete documents.

Additionally, we used the generated specifications to check the API documentation and found issues within the documentation itself, which can mislead developers and cause serious security impacts. Specifically, we identified three types of issues in the documentation: *(1) Outdated document*: The API documentation was not updated after the API code was modified, leading to outdated information. For example, the API `backlight_device_get_by_name` document mentioned the post-operation `backlight_put`, which has been deprecated. The correct post-operation should be `put_device`. *(2) Typo in document*: The API documentation contained typo errors for post-operation. For instance, the `enclosure_find` API should use `put_device` for release, but the documentation incorrectly stated `device_put`. *(3) Error due to copy-paste*: Developers may reference similar APIs while writing the documentation but fail to notice differences in post-operations, leading to errors in the API specifications described. For example, `phy_put` corresponds to `phy_get`, but `of_phy_get` does not correspond to `phy_put`. Figure 13 shows the documentation for the `of_phy_get` API, which incorrectly states that the caller should use `phy_put` to release the count. However, the correct function is `of_phy_put`. Unlike `of_phy_put`, `phy_put` also calls the `device_link_remove` function to delete a stateless link between two devices. If developers follow the documentation and mistakenly use `phy_put` to release the count obtained by `of_phy_get`, it may result

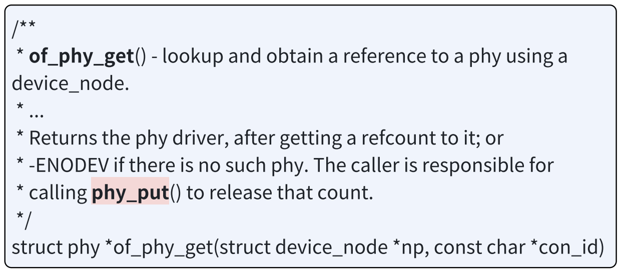

```
/**
 * of_phy_get() - lookup and obtain a reference to a phy using a
device_node.
 * ...
 * Returns the phy driver, after getting a refcount to it; or
 * -ENODEV if there is no such phy. The caller is responsible for
 * calling phy_put() to release that count.
 */
struct phy *of_phy_get(struct device_node *np, const char *con_id)
```

Fig. 13: Error in the document of `of_phy_get`

in the device being unregistered while still in use, leading to Use-After-Free or other unexpected behaviors. APISpecGen successfully generated the correct post-operations for the APIs, thus helping to identify the documentation errors. We have submitted patches to fix these documentation errors, which have been merged into the Linux kernel's main branch.

**The Frequency of Correct API Usage.** We analyzed the frequency of correct API usage for the generated specifications to determine whether they meet the occurrence thresholds set by existing methods. Following previous work [3], [40], we performed intra-function analysis to count how many API callers adhere to each API specification. Typically, previous studies [12], [3] set a threshold of 10 for frequent pattern mining, meaning that at least 10 callers must follow the API usage specification for mining. However, the results show that over 93% of API pairs occur less than 10 times. Additionally, 84% of API pairs appear less than 5 times. This indicates that frequent pattern mining methods overlook the majority of APIs specifications due to insufficient correct usage. As API design becomes more specialized and involves multiple layers of abstraction, the number of calls to an API can be very small. APISpecGen does not rely on the frequent occurrence of correct API usage. It generates specifications through specification propagation analysis, thus covering APIs that are not frequently used.

**The Characteristic of API Name.** API names often contain semantic clues, and several existing methods use them to identify specific API types, such as those related to memory operations. In particular, for paired function specifications, API names often include subwords like *"alloc"*, *"new"*, or *"request"*, indicating resource allocation. These APIs, therefore, typically require post-operations to release the allocated resources. We analyzed the function pairs in the generated specifications and identified subwords that frequently appear in API names, like *"get"*,*"find"*,*"alloc"*,*"lookup"*,*"request"*, and *"new"*. However, we also observed that 12.71% of APIs do not contain these informative subwords, meaning their names do not explicitly suggest resource allocation, even though they may still require post-operations. In general, lower-level APIs often have names that clearly indicate resource management, such as `get_device` and `put_device`. However, as APIs evolve to handle more complex or varied use cases, their names may no longer directly indicate resource management. For instance, the API `of_path_to_platform_device` does not suggest any resource management, but in reality, it acquires a reference and requires the `platform_device_put` function to release it. Using `get_device` as a seed, APISpecGen, traces the specification propagation path from `get_device` to more specific APIs like `bus_find_device`, `bus_find_device_by_of_node`, and finally to `of_path_to_platform_device`. The results show that some APIs' names lack obvious semantics to indicate the need for post-operations. Relying on API names may miss these APIs' specifications. APISpecGen does not rely on API name and can generate specifications for these APIs.

## E. Findings

Based on the results, we share our findings about the API design and their specifications.

**(1) APIs often evolve from primitive APIs and share similar specifications.** Our results reveal that many APIs share similar specifications, a relationship that is typically established during the design phase. Specifically, some APIs are derived from basic APIs. As APIs evolve, they adapt to different scenarios, and their corresponding post-operations adapt accordingly. Despite these variations, the underlying relationships between the APIs and their post-operations remain similar. For example, as shown in Table V, function pairs like `scsi_host_get` and `scsi_host_put` follow a pattern similar to that of primitive APIs `get_device` and `put_device`. These function pairs can be traced back to primitive APIs.

TABLE V: Specification derived from get_device

| Target API | Post-Operation |
|---|---|
| _zfcp_unit_find | put_device |
| scsi_host_get | scsi_host_put |
| rproc_get_by_phandle | rproc_put |
| cxl_afu_get | cxl_afu_put |
| regulator_get | regulator_put |
| pci_p2pmem_find | pci_dev_put |
| wpan_phy_find | wpan_phy_put |

**(2) APIs are designed at different levels for convenience, and they require consistent specifications.** When developing software, developers often work with APIs that span multiple levels and serve various use cases. For instance, in the Linux kernel's device driver infrastructure, the `get_device` API is utilized across different layers, with multiple variations that offer different ways of obtaining devices. These APIs are designed for diverse use cases, and they still require consistent specifications to ensure correct usage. Table VI illustrates some of these variations and their hierarchical relationships. At the first level, we have functions like `bus_find_device`, `driver_find_device`, and `class_find_device`. While all of these functions are based on `get_device` to obtain devices, each is designed for a specific use case—such as working with buses, drivers, or specific device types. At the second level, APIs like `driver_find_device_by_name` and `driver_find_device_by_fwnode` offer more specific ways to obtain devices. These variations give developers flexibility in obtaining device references at different levels of abstraction. They are widely used across various drivers and often misused because developers overlook the specifications.

TABLE VI: Two API layers in kernel device infrastructure

| Level-1 API | Level-2 API |
|---|---|
| driver_find_device | driver_find_device_by_name |
| | driver_find_device_by_fwnode |
| | chsc_get_next_subchannel |
| bus_find_device | bus_find_device_by_acpi_dev |
| | bus_find_device_by_fwnode |
| | wmi_find_device_by_guid |
| class_find_device | fpga_region_class_find |
| | class_find_device_by_name |
| | regulator_lookup_by_name |

## VIII. Discussion

**The Scope and Extension.** In this paper, we focus on improper API post-handling, which accounts for 66% of API misuse [8], leading to various bug types (e.g., memory corruption, refcount imbalance, and error handling issues) and severe security impacts (e.g., Use-After-Free, system crash) [6], [25], [31], [38]. APISpecGen can be extended to other patterns of API specifications (e.g., checking API parameters) with minimal effort [41], [22]. Most API specifications share common elements—APIs, critical variables, and security operations—that differ mainly in their order or operation type. To extend APISpecGen, we only need to simply adjust propagation models (as shown in Figure 2) and analysis. Our critical variable-based propagation flow analysis can easily adapt to other patterns.

Similar to other static analysis-based studies [20], [17], APISpecGen has inherent limitations. Static analysis often struggles with indirect calls and inter-procedural analysis, and its accuracy can be difficult to guarantee [18]. Despite this, static analysis is efficient for large-scale code analysis, and APISpecGen performs well in generating specifications and detecting bugs. To address the limitations of static analysis, advanced techniques like indirect call analysis can be integrated to achieve more accurate inter-procedural analysis [19].

**Object Propagation Analysis in Bug Detection.** Some bug detectors use object propagation analysis to track objects allocated by low-level APIs like `kmalloc` and identify necessary release points [6]. However, starting from primitive APIs is inefficient. Program complexity often causes data flow explosions, limiting analysis to low-level calls and making it difficult to trace higher-level APIs. Additionally, some bug detectors use escape analysis to reduce false positives by identifying whether a variable escapes a function [31]. But this approach cannot confirm if escaped objects are properly managed. In contrast, APISpecGen uses specification propagation analysis to generate API specifications, bypassing the need for bottom-up tracing in bug detection. In this way, we can focus only on the specific API usage, and improves the effectiveness of bug detection.

## IX. Related Work

**API Misuse Detection.** API misuse detection on focus on extracting API specifications and identifying their violations to detect misuse. Existing methods extract API specifications from various artifacts, including API documentation [24], [30], [42], [28], API usage code [12], [20], [23], [14], [16], and bug patches [17]. For example, Advance [24] extracts specifications by identifying strong sentiment sentences in the documentation. APP-Miner [12], APISan [40], and Crix [23] infer specifications by analyzing frequent API usage patterns. APHP [17] derives specifications from patched code and patch descriptions. APICAD [36] and AURC [10] combine multiple sources, including documentation, API usage code, and API source code. Unlike methods that rely on external artifacts for specification extraction, APISpecGen leverages the inherent relationships between APIs for new specification generation, thus can detect a wider range of API misuses.

**Specialized Bug Detection.** Some studies focus on detecting specific types of bugs caused by API misuse, such as memory leaks, memory corruption, and reference count imbalance [6], [25], [31], [38]. These methods define key API characteristics for particular bug types and identify APIs that match these characteristics to detect misuse. For example, CID [31] focuses on reference count leaks by identifying APIs that increment reference counters and checking for corresponding decrement operations through consistency analysis. HERO [38] focuses on error-handling bugs and extracts function pairs related to error handling. K-Meld [6] detects memory leaks by identifying memory allocation APIs and analyzing their usage patterns. Similarly, Goshark [25] uses natural language processing (NLP) and data flow analysis to identify memory-related functions. Unlike methods that focus on specific bug types, APISpecGen generates specifications for diverse APIs, thus can detect diverse misuses across various bug types.

## X. Conclusion

In this paper, we introduce API specification propagation and present APISpecGen, a framework that generates new API specifications from existing ones. Using seed specifications, APISpecGen iteratively performs bidirectional propagation analysis and combines API usage with data-flow validation to ensure accurate specification generation. Experimental results show that using 6 seed specifications, APISpecGen generated 7332 new API specifications. Using the generated by APISpecGen, we detected 186 unknown bugs in the Linux kernel, with 8 CVEs assigned.

## Acknowledgments

We would like to thank our reviewers for their insightful comments. We are also grateful to the maintainers of open source software for their invaluable feedback. The IIE authors are supported in part by NSFC (U24A20236, 92270204), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), Youth Innovation Promotion Association CAS and the Open Research Fund of Joint Laboratory on Cyberspace Security, China Southern Power Grid (Grant No. GDKJXM20230642).

## References

[1] "Linux kernel commit, d46fe2cb2dc." https://github.com/torvalds/linux/commit/d46fe2cb2dc, 2024.

[2] M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, "Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code," in *Proceedings of the 30st USENIX Security Symposium (Security)*, 2021.

[3] P. Bian, B. Liang, J. Huang, W. Shi, X. Wang, and J. Zhang, "SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed," in *Proceedings of the 28th SIGSOFT Foundations of Software Engineering (FSE)*, 2020.

[4] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai, "Nar-miner: Discovering negative association rules from code for bug detection," in *Proceedings of the 26th SIGSOFT Foundations of Software Engineering (FSE)*, 2018.

[5] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai, "Detecting bugs by discovering expectations and their violations," *IEEE Transactions on Software Engineering*, 2018.

[6] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning," in *Proceedings of the 28th Network and Distributed System Security Symposium (NDSS)*, 2021.

[7] FFmpeg, "FFmpeg," https://git.ffmpeg.org/ffmpeg.git, 2024.

[8] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu, "An Empirical Study on API-Misuse Bugs in Open-Source C Programs," in *Proceedings of the 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 2019.

[9] L. He, H. Hu, P. Su, Y. Cai, and Z. Liang, "Freewill: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries," in *USENIX Security Symposium*, 2022.

[10] P. Hu, R. Liang, Y. Cao, K. Chen, and R. Zhang, "AURC: Detecting errors in program code and documentation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[11] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically Detecting Error Handling Bugs Using Error Specifications," in *Proceedings of the 25th USENIX Security Symposium (Security)*, 2016.

[12] J. Jiang, J. Wu, X. Ling, T. Luo, S. Qu, and Y. Wu, "App-miner: Detecting api misuses via automatically mining api path patterns," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[13] Y. J. Kang, B. Ray, and S. S. Jana, "APEx: Automated Inference of Error Specifications for C APIs," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, 2016.

[14] Z. Li and Y. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code," in *Proceedings of the 13th SIGSOFT Foundations of Software Engineering (FSE)*, 2005.

[15] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, "ARBITRAR: User-Guided API Misuse Detection," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, 2021.

[16] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "AntMiner: Mining More Bugs by Reducing Noise Interference," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[17] M. Lin, K. Chen, and Y. Xiao, "Detecting API Post-Handling bugs using code and description in patches," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[18] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[19] D. Liu, S. Ji, K. Lu, and Q. He, "Improving indirect-call analysis in llvm with type and data-flow co-analysis," in *USENIX Security Symposium*, 2024.

[20] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen, and Q. He, "Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths," *In Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, 2021.

[21] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution," in *Proceedings of the 31st USENIX Security Symposium (Security)*, 2022.

[22] J. Liu, Y. Yang, K. Chen, and M. Lin, "Generating api parameter security rules with llm for api misuse detection," *ArXiv*, vol. abs/2409.09288, 2024.

[23] K. Lu, A. Pakki, and Q. Wu, "Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences," in *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[24] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, 2020.

[25] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu, and J. Li, "Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 2096–2113.

[26] Z. Mousavi, C. Islam, M. A. Babar, A. Abuadbba, and K. Moore, "Detecting misuse of security apis: A systematic review," 2023.

[27] openssl, "openssl," https://github.com/openssl/openssl, 2024.

[28] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring Method Specifications from Natural Language API Descriptions," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[29] M. K. Ramanathan, A. Y. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2007.

[30] X. Ren, X. Ye, Z. Xing, X. Xia, X. Xu, L. Zhu, and J. Sun, "API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph," in *Proceedings of the 35th International Conference on Automated Software Engineering (ICSE)*, 2020.

[31] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking," in *Proceedings of the 30th USENIX Security Symposium (Security)*, 2021.

[32] R. Tartler, D. Lohmann, C. J. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," in *Programming Languages and Operating Systems*, 2011.

[33] G. D. Team, "GLib Documentation," https://docs.gtk.org/glib/, 2024.

[34] tree sitter, "tree-sitter," https://tree-sitter.github.io/tree-sitter/, 2024.

[35] J. Wang, S. Ma, Y. Zhang, J. Li, Z. Ma, L. Mai, T. Chen, and D. Gu, "Nlp-eye: Detecting memory corruptions via semantic-aware memory operation function identification," in *International Symposium on Recent Advances in Intrusion Detection*, 2019.

[36] X. Wang and L. Zhao, "Apicad: Augmenting api misuse detection through specifications from code and documents," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[37] weggli rs, "weggli," https://github.com/weggli-rs/weggli, 2024.

[38] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[39] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, 2014.

[40] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API Usages through Semantic Cross-Checking," in *Proceedings of the 25th USENIX Security Symposium (Security)*, 2016.

[41] H. Zhong, N. Meng, Z. Li, and L. Jia, "An empirical study on api parameter rules," *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 899–911, 2020.

[42] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs Documentation and Code to Detect Directive Defects," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017.

TABLE VII: List of 60 bugs in Linux kernel detected with the specifications generated by APISpecGen. Each row displays the buggy function, target API, post-operation, and seed API for specification generation and security impact.

| Buggy Function | Target API | Post-Operation | Seed API | Security Impact |
|---|---|---|---|---|
| omapdss_init_of | omapdss_find_dss_of_node | of_node_put | get_device | refcount leak |
| ti_sci_intr_irq_domain_probe | of_irq_find_parent | of_node_put | get_device | refcount leak |
| msc313_gpio_probe | of_irq_find_parent | of_node_put | get_device | refcount leak |
| mvebu_gicp_probe | of_irq_find_parent | of_node_put | get_device | refcount leak |
| alpine_msix_init_domains | of_irq_find_parent | of_node_put | get_device | refcount leak |
| sifive_gpio_probe | of_irq_find_parent | of_node_put | get_device | refcount leak |
| stm32_pctrl_get_irq_domain | of_irq_find_parent | of_node_put | get_device | refcount leak |
| ixp4xx_gpio_probe | of_irq_find_parent | of_node_put | get_device | refcount leak |
| ti_sci_intr_alloc_parent_irq | of_irq_find_parent | of_node_put | get_device | refcount leak |
| platform_irqchip_probe | of_irq_find_parent | of_node_put | get_device | refcount leak |
| ti_sci_inta_irq_domain_probe | of_irq_find_parent | of_node_put | get_device | refcount leak |
| rpi_ts_probe | rpi_firmware_get | rpi_firmware_put | get_device | refcount leak |
| vc4_hvs_bind | rpi_firmware_get | rpi_firmware_put | get_device | refcount leak |
| hns_roce_mmap | rdma_user_mmap_entry_get_pgoff | rdma_user_mmap_entry_put | get_device | refcount leak |
| erdma_mmap | rdma_user_mmap_entry_get | rdma_user_mmap_entry_put | get_device | refcount leak |
| bwmon_probe | dev_pm_opp_find_bw_ceil | dev_pm_opp_put | get_device | refcount leak |
| __dtpm_cpu_setup | cpufreq_cpu_get | cpufreq_cpu_put | get_device | refcount leak |
| hns_mac_register_phy | hns_dsaf_find_platform_device | put_device | get_device | refcount leak |
| x86_instantiate_serdev | device_find_child_by_name | put_device | get_device | refcount leak |
| dpaa2_mac_connect | of_phy_get | of_phy_put | get_device | refcount leak |
| fme_bridge_enable_set | dfl_fpga_cdev_find_port | put_device | get_device | refcount leak |
| fdp1_probe | rcar_fcp_get | rcar_fcp_put | get_device | refcount leak |
| of_led_get | class_find_device_by_of_node | put_device | get_device | refcount leak |
| nfc_genl_vendor_cmd | nfc_get_device | put_device | get_device | refcount leak |
| nfc_genl_se_io | nfc_get_device | put_device | get_device | refcount leak |
| bcm_sf2_mdio_register | of_mdio_find_bus | put_device | get_device | refcount leak |
| usb_otg_start | usb_get_phy | usb_put_phy | get_device | refcount leak |
| pxa_udc_probe | usb_get_phy | usb_put_phy | get_device | refcount leak |
| xgmiitorgmii_probe | of_phy_find_device | put_device | get_device | refcount leak |
| amd8132_probe | pci_dev_get | pci_dev_put | get_device | refcount leak |
| ubi_detach_mtd_dev | ubi_get_device | put_device | get_device | refcount leak |
| am33xx_pm_probe | wkup_m3_ipc_get | wkup_m3_ipc_put | get_device | refcount leak |
| bebob_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| motu_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| efw_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| oxfw_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| snd_ff_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| dice_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| snd_dg00x_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| snd_tscm_probe | fw_unit_get | fw_unit_put | get_device | refcount leak |
| nvram_init | get_mtd_device_nm | put_mtd_device | get_device | refcount leak |
| kfr2r09_usb0_gadget_i2c_setup | i2c_get_adapter | i2c_put_adapter | get_device | refcount leak |
| vpif_probe | i2c_get_adapter | i2c_put_adapter | get_device | refcount leak |
| rockchip_init_usb_uart | of_find_matching_node_and_match | of_node_put | get_device | refcount leak |
| mvebu_mbus_dt_init | of_find_matching_node_and_match | of_node_put | get_device | refcount leak |
| psci_dt_init | of_find_matching_node_and_match | of_node_put | get_device | refcount leak |
| bq25890_fw_probe | power_supply_get_by_name | power_supply_put | get_device | refcount leak |
| iscsi_set_host_param | scsi_host_lookup | scsi_host_put | get_device | refcount leak |
| spm_cpuidle_register | of_cpu_device_node_get | of_node_put | get_device | refcount leak |
| parse_perf_domain | of_cpu_device_node_get | of_node_put | get_device | refcount leak |
| sii8620_init_rcp_input_dev | rc_allocate_device | rc_free_device | device_initialize | refcount leak |
| highbank_mc_probe | edac_mc_alloc | edac_mc_free | device_initialize | refcount leak |
| mmc_omap_new_slot | mmc_alloc_host | mmc_free_host | device_initialize | refcount leak |
| fme_bridge_enable_set | dfl_fpga_port_ops_get | dfl_fpga_port_ops_put | try_module_get | refcount leak |
| cs35l41_hda_read_acpi | acpi_get_subsystem_id | kfree | kstrdup | memory leak |
| __tegra_channel_try_format | __v4l2_subdev_state_alloc | __v4l2_subdev_state_free | kmalloc | memory leak |
| btmtksdio_probe | hci_alloc_dev | hci_free_dev | kmalloc | memory leak |
| davinci_mdio_probe | alloc_mdio_bitbang | free_mdio_bitbang | kmalloc | memory leak |
| xen_pcibk_xenbus_probe | alloc_pdev | free_pdev | kmalloc | memory leak |
| efx_tc_flower_handle_lhs_actions | efx_tc_get_recirc_id | efx_tc_put_recirc_id | kmalloc | memory leak |
| highbank_l2_err_probe | edac_device_alloc_ctl_info | edac_device_free_ctl_info | kmalloc | memory leak |
| mt798x_phy_calibration | nvmem_cell_get | nvmem_cell_put | kmalloc | memory leak |
| virtio_gpu_object_shmem_init | drm_gem_shmem_get_sg_table | IS_ERR | ERR_PTR | null-ptr-deference |
| malidp_check_pages_threshold | get_sg_table | IS_ERR | ERR_PTR | null-ptr-deference |
| q6apm_graph_open | gpr_alloc_port | IS_ERR | ERR_PTR | null-ptr-deference |
| ufs_mtk_init_va09_pwr_ctrl | regulator_get | IS_ERR | ERR_PTR | null-ptr-deference |
| imx8mq_soc_revision | of_clk_get_by_name | IS_ERR | ERR_PTR | null-ptr-deference |
| expr__ctx_new | hashmap__new | IS_ERR | ERR_PTR | null-ptr-deference |
| dr_domain_init_resources | mlx5_get_uars_page | IS_ERR | ERR_PTR | null-ptr-deference |
| memory_tier_init | alloc_memory_type | IS_ERR | ERR_PTR | null-ptr-deference |

## Appendix A
## Artifact Appendix

### A. Description & Requirements

*1) How to access:* We provide public access to our code and experiment setups through the following Zenodo link:
https://doi.org/10.5281/zenodo.14244150
You can also access it in Github:
https://github.com/Yuuoniy/APISpecGen
The artifact is licensed under Apache License 2.0.

*2) Hardware dependencies:* In our evaluation, we used a 64-bit Ubuntu 22.04 system with 503GB of memory and 5TB of storage, powered by an Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz with 79 processors.

APISpecGen works on standard machines. It is implemented in a multithreaded manner, and we recommend using a multithreaded computer to speed up the evaluation process.

*3) Software dependencies:* APISpecGen is implemented in Python and leverages existing code analysis tools, including Joern, Tree-sitter, and Weggli. We provide a Dockerfile to automate the setup and creation of the Docker environment. This Dockerfile includes everything needed to configure the runtime environment, install third-party tools and dependencies, set up Python libraries.

*4) Benchmarks:* Source code of the Linux kernel (e.g., linux-5.16).

### B. Artifact Installation & Configuration

This section should include all the high-level installation and configuration steps required to prepare the environment to be used for the evaluation of your artifact.

### C. Experiment Workflow

The main workflow involves using given specification seeds to generate new API specifications, and using these generated specifications to detect new bugs. Therefore, the experiemts include two main stage: specifications generation and bug detection.

*1) Installation:* We provide Dockerfile to simplify the installation. Users can build the docker for APISpecGen using the *Dockerfile* and download the program for testing using the commands below.

```
$ wget https://github.com/Yuuoniy/
APISpecGen/raw/refs/heads/main/Dockerfile
$ docker build -t apispecgen:latest .
$ docker run -it --name "apispecgen"
"apispecgen:latest"
```

*2) Basic Test:* we provide a minimal running example for quick test. This can reproduce the working example we introduced in the paper (Figure 5). Please run the command as follows. This test generates the specification for API `nfc_get_device` and detects the buggy function `nfc_genl_vendor_cmd`.

1. [5-minutes] Basic test for specification generation.
```
$./script/0.quick_spec_generate.sh
```
2. [2-minutes] Basic test for bug detection.
```
$./script/0.1.quick_bug_detection.sh
```

### D. Major Claims

- (C1): APISpecGen extracts thousands API specifications using given six seeds. This is proven by the experiments (E1) in the Section VII.A, whose results are illustrated in Table II.
- (C2): Using the generated specifications, APISpecGen detects numerous new bugs in the Linux Kernel, This is proven by the experiments (E2) in the Section VII.B.
- (C3) Most of the generated specifications cannot be extract with the API artifacts. This is proven by the experiments (E3) in the Section VII.D.

### E. Evaluation

*1) (E1):* [Specification Generation] [5 compute-hour]:
Generate specifications use the given six seed specifications in the Linux Kernel.

*[Preparation]* Follow the previously instructions for installation and downloads the source code of linux kernel. Run the basic test to check the environment. Make sure the `config.cfg` correctly setup the path for source code.

*[Execution]* Please run:
```
$./script/1.specification_generation.sh
```
*[Results]* Generated specifications are saved in DIR `SpecGeneration/Data/GeneratedSpec`. The default max_depth is 10. The specifications follow the defined three-tuple format and including the propagation information.

*2) (E2):* [Bug Detection] [5 compute-hour]:
Use generated specifications to detect new bugs in the Linux kernel. To facilitates evaluation, we provided a set of specifications that related to detected bugs for validation. All the used specifications are previously generated.

*[Preparation]* Run the basic test for bug detection to check the environment.

*[Execution]* Please run
```
$./script/2.bug_detection.sh
```
*[Results]* Using the specifications, APISpecGen detects hundreds of new bugs in the Linux kernel. The bugs reports is saved to file `BugDetection/data/bug_report.csv`.

*3) (E3):* [API Artifacts Evaluation] [5 minutes]:
Use the generated specifications to evaluate the usability of API artifacts (including API documentation, API names, and API usage) for specification extraction.

*[Preparation]* In the evaluation, we provide the previously generated specifications for analysis. Alternatively, users can specify the `spec_file` in `APIAritifactEval/APIAritifactEval.py`.

*[Execution]* Please run:
```
$./script/3.API_aritifact_analysis.sh
```
*[Results]* This prints out the statistical data which reveals that API artifacts have significant limitations in specification extraction.