# Balancing Analysis Time and Bug Detection: Daily Development-friendly Bug Detection in Linux

Keita Suzuki, *Keio University;* Kenta Ishiguro, *Hosei University;* Kenji Kono, *Keio University*

https://www.usenix.org/conference/atc24/presentation/suzuki

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

# Balancing Analysis Time and Bug Detection: Daily Development-friendly Bug Detection in Linux

Keita Suzuki [*]
*Keio University*

Kenta Ishiguro
*Hosei University*

Kenji Kono
*Keio University*

## Abstract

Linux, a battle-tested codebase, is known to suffer from many bugs despite its extensive testing mechanisms. While many of these bugs require domain-specific knowledge for detection, a significant portion matches well-known bug patterns. Even though these bugs can be found with existing tools, our simple check of Linux kernel patches suggests that these tools are not used much in the developer's daily workflow. The lack of usage is probably due to the well-known trade-off between analysis time and bug detection capabilities: tools typically employ complex analysis to effectively and comprehensively find bugs in return for a long analysis time, or focus on a short analysis time by only employing elementary analyses and thus can only find a very limited number of bugs. Ideally, developers expect the tools to incur short analysis time, while still finding many bugs to use them in daily development.

This paper explores an approach that balances this trade-off by focusing on bugs that can be found with less computationally-complex analysis methods, and limiting the scope to each source code. To achieve this, we propose a combination of computationally lightweight analyses and demonstrate our claim by designing FiTx, a framework for generating daily development-friendly bug checkers that focus on well-known patterns. Despite its simplicity, FiTx successfully identified 47 new bugs in the Linux kernel version 5.15 within 2.5 hours, outperforming Clang Static Analyzer and CppCheck in both speed and bug detection. It demonstrates that focusing on less complex bug patterns can still significantly contribute to the improvement of codebase health. FiTx can be embedded into the daily development routine, enabling early bug detection without sacrificing developers' time.

## 1 Introduction

Linux, a popularly used operating system, is one of the most sophisticated system software with many well-experienced developers contributing to the codebase. In 2021, 86,023 commits were made in the year alone (average of more than 200 commits per day), with over 4,500 developers contributing to the codebase of over 30 million lines of code as of version 5.15. Unfortunately, the Linux kernel contains many bugs, and there has been much work that reported various bugs [9–14, 17, 18, 24, 25, 27, 28, 30, 33, 34, 38, 39, 43, 49, 50, 53, 54, 58, 59].

A substantial number of these bugs are known to align with well-known bug patterns, allowing them to be detected without domain-specific knowledge. These bugs often manifest themselves repeatedly within the codebase across a wide variety of projects. For instance, the survey conducted by Bai et al. [9] found 949 commits in Linux fixing use-after-free in the years 2016 to 2018, and Li et al. [32] found 365 null-pointer dereferences in Linux version 5.6. Finding these bugs is important since even a simple memory bug can cause severe damage such as a system crash.

Despite extensive efforts by kernel developers to address well-known bug patterns with diverse methods including fuzzing, integration testing, or static code analysis, the use of bug detection tools appears to be less than expected. Tools such as Kernel Address Sanitizer (KASAN) [5] or Clang Static Analyzer (CSA) [6] have contributed greatly to the detection of memory bugs such as double-free or use-after-free. However, our simple check of Linux kernel bug-fixing patches shows that only 37.5% (24 out of 64 patches) of the patches mentioned any use of tools, even though it is strongly recommended to credit the tools.

The limited integration of bug detection tools into developer's daily development workflows is most likely for the trade-off between analysis time and bug detection capabilities. Tools that find many bugs comprehensively require a long analysis time. For instance, PATA [32] conducts a path-sensitive, interprocedural, alias-aware analysis and can find 574 bugs in Linux, but it required 33 hours of analysis time because of the complex analysis techniques it leverages. Conversely, tools prioritizing short analysis time, such as CSA or CppCheck [19], sacrifice bug detection capabilities and suffer from many false positives. CppCheck only conducts path-

[*]now at Google

insensitive analysis and does not conduct inter-procedural analysis on pointer values, hence it only requires 2 hours and 32 minutes of analysis time but does not target many bug patterns that Linux suffers.

During daily development, developers prioritize rapid feedback from bug-detection tools to minimize interference to their development. This drives them to demand for short analysis time and use tools such as CSA, which is readily integrated into Linux Makefile. However, the challenge remains for tools prioritizing short analysis to maximize their bug detection capabilities while maintaining the analysis time.

In this paper, we aim to explore a balanced approach that finds bugs while still maintaining developers' daily development throughput, and present one promising combination of analyses to detect well-known bug patterns. We tackle the traditional trade-off of analysis time and bug detection capabilities, and deal with the challenge of short analysis time while finding bugs by achieving two goals:

**Short analysis time.** The approach prioritizes short analysis time by limiting the analysis to less computationally complex ones, and by focusing on individual source files. Our prototype only required 0.99 seconds for 90% of the analyzed source files. This enables seamless integration with developers' daily workflow without compromising productivity.

**Targeted bug detection.** Despite only targeting the bugs findable with efficient, computationally simple approaches, our combination of the low-cost analyses can still present positive and meaningful bug-detection results to the developers. Our prototype was able to find 47 new bugs in Linux v5.15.

Our contribution is to present a combination of computationally low-cost analyses that can impactfully find bugs in Linux. In detail, the leveraged analyses are as follows: (1) only inspects a *single compilation unit* (translation unit or single file in C), (2) deals with *static field offsets* that do not require runtime calculation of field offsets, meaning it only deals with struct fields or hard-coded array indexes. (3) only conducts *lightweight alias analysis* involving *intra*-procedural and path-*insensitive* analysis, and (4) does *not consider indirect function calls*.

We implement the approach as FiTx, a framework to generate daily development-friendly bug checkers. FiTx leverages typestate analysis [20, 22, 26, 32, 41, 52, 56] to efficiently analyze well-known bug patterns, and conduct a path-insensitive and inter-procedural analysis of the control flow graph [1]. The inter-procedural analysis uses a bottom-up, summary-based approach to achieve a short analysis time. To achieve greater accuracy with the summary-based analysis, we introduce a technique called *return-code aware state-propagation*. It summarizes the states for each return code of each callee function and propagates the appropriate state to each path in the callee.

Surprisingly, FiTx has detected 47 new bugs in Linux kernel version 5.15 with 2 hours 33 minutes of analysis time to analyze the entire Linux. Our tool incurs less than 0.20 seconds of analysis time for 50% of the source files, and 0.99 seconds for 90% of the source files. As of the time of writing, 13 of the bugs are confirmed by the Linux developers [42, 44–48]. This included files such as `trace_events_hist.c` in the `kernel` directory which is one of the core directories with many developers contributing to the source file.

[1]Available at `https://github.com/sslab-keio/FiTx`

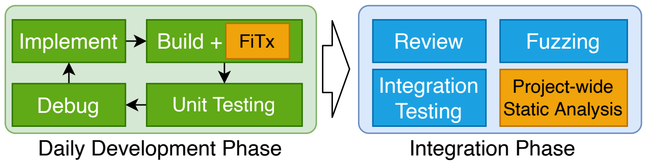


Figure 1: Development cycle with FiTx

Figure 1 shows the development flow with FiTx. FiTx generates compiler extensions to seamlessly integrate into the development process, acting as an early warning system for potential bugs. It analyzes source code on each build, promptly notifying developers of likely issues for immediate attention. FiTx is not intended to replace "sophisticated" comprehensive static analysis tools. Instead, FiTx complements these tools by addressing early-stage bugs, allowing more sophisticated tools to focus on intricate issues during later phases like integration.

FiTx can find more bugs with a short analysis time compared to CSA and CppCheck. Neither tool found the 13 developer-confirmed bugs FiTx found. CSA's path-sensitive approach for lower false positive rates necessitates the need for strong limitations in inter-procedural analysis to shorten analysis time. This overlooks many bugs. It required 10.74 seconds for 90% of the source files. CppCheck focuses on finding environment-specific bugs by checking multiple macro configurations with basic analyses. Although they require 0.32 seconds for 90% of the source files, it overlooks many bugs due to the lack of inter-procedural analysis on pointer values.

FiTx effectively minimizes the effort required for a false positive filtering, a common challenge in bug detection. While most tools tackle this issue by lowering the false positive rate, FiTx deals with it by generating a substantially low number of warnings, reducing the manual check required by the developers. Our prototype generated 113 warnings with 66 false positives. For 99.9% of source files, the developer needed to check at most three warnings per source file. This is significantly fewer than CSA and CppCheck which generated 132,196 and 1,528 warnings respectively on default configuration. In addition, FiTx generates state transition logs to aid the developers in checking the warnings. On average, it required less than two minutes per warning for an inexperienced Ph.D. student without any domain knowledge to determine them.

The rest of this paper is organized as follows: Section 2 motivates our work by characterizing bug detection tools in Linux, and Section 3 explains the goal of this paper. Sections 4 and 5 show the design and implementation. Section 6 gives an evaluation of our approach, and Section 7 discusses the limitations. Section 8 relates our work with others and Section 9 concludes this paper.

## 2 Bug Detection Tools in Linux Kernel

The Linux kernel, despite employing a rigorous code-checking process including code review, integration testing, or static/dynamic analysis, is known to still suffer from many bugs. Although many of these bugs require domain-specific knowledge to find them, a substantial amount of bugs fulfill well-known bug patterns, and static bug detection tools have been contributed greatly to finding many of these bugs in Linux [9–14, 17, 18, 24, 25, 27, 28, 30, 33, 34, 38, 39, 43, 49, 50, 53, 54, 58, 59]. For instance, PATA [32] found 454 new bugs in the Linux kernel for bug patterns such as null-pointer dereference, uninitialized variable access, and memory leaks.

Despite being effective, these bug detection tools often remain absent in the daily development of the developers. According to a simple check of Linux kernel bug-fixing patches, only 24 out of 64 patches (37.5%) mentioned the use of bug detection tools (Details in Section 3.1.2). It is customary for developers to credit the bug detection tools in their patch message, so the lack of appearance suggests that these bugs are found with alternative methods, including manual discovery by the developer without the help of these tools.

The underutilization of static bug detection tools in the developers' daily workflows is possibly due to the well-known trade-off between analysis time and bug detection capabilities. Tools emphasizing detailed analysis can find many bugs, but at the cost of lengthy execution times due to their complex analysis methods. These tools are fit for the later phase of development such as integration, where developers take time to evaluate their code. Contrary, tools prioritizing speed often miss numerous bugs common in Linux due to their limited analysis methods. Developers ideally want to identify many impactful bugs while minimizing development delays, hence value these tools with short analysis time in daily workflows. However, the lack of bug-detection capabilities of these tools makes them less attractive to developers for practical use.

Table 1 shows the leveraged analysis methods and the analysis time of well-known tools when analyzing Linux kernel (v5.6 for PATA, v5.15 for the others) configured with `allyesconfig`. The static bug detection tools are Clang Static Analyzer (CSA, v10.0.1) [6], Cppcheck (v1.90) [19], Coccinelle (v1.1.1) [36], Saber (v2.1) [57], and PATA [32]. For CSA, CppCheck, and Saber we run the tools with the default settings, or the settings already provided in the Linux for use. For Coccinelle, we use the semantic patch for NULL pointer dereference provided in the Linux kernel. Since PATA is not publicly available, we use the results reported in their paper [32]. The analysis is run on the setup shown in Table 7.

PATA and Saber are both standalone tools that conduct comprehensive and complex analyses such as path-sensitive data flow analysis or interprocedural alias analysis. PATA requires more than 33 hours to finish the analysis. Saber cannot finish the analysis due to a lack of memory (32 GB).

CSA conducts complex analysis (path-sensitive data flow analysis with alias analysis). Although CSA is configured to analyze a single compilation unit, it requires 10.75 seconds for 90% of the source files. CSA is readily integrated into Linux Makefile. However, due to its strong limitations with interprocedural analysis (limits to 1 call depth) and the number of paths to explore, it misses many bugs. CSA can run during compilation when using the Clang compiler [1].

Unlike previous tools, CppCheck and Coccinelle leverage fairly simple analysis on a single compilation unit. Both tools run independently from the build process. CppCheck conducts path-insensitive dataflow analysis without alias information, and does not conduct interprocedural analysis of pointer variables. CppCheck requires 0.32 seconds for 90% of the source files. Coccinelle transforms the given code using developer-defined transformation rules (semantic patches), but can also be used to find certain violations of code patterns. However, it does not conduct interprocedural dataflow analysis [36, 40]. It requires 0.81 seconds for 90% of the source files.

## 3 Goal and Key Observation

In this paper, we address the challenge of effectively finding bugs while minimizing interruptions to developers' daily workflows, and propose one promising combination of analysis techniques that can be leveraged to tackle the trade-off between analysis time and bug detection capabilities. Our approach leverages a carefully chosen combination of less computationally complex analyses with focused analysis scopes, and targets well-known bug patterns findable with these analyses. It allows to achieve short analysis time by avoiding complex analysis, allowing for seamless integration into developers' daily workflows without sacrificing bug detection capabilities.

In detail, the proposed combination of analysis is as follows: the analysis scope is a *single* compilation unit; the dataflow analysis is *inter*-procedural, path-*insensitive*, and *field-based* with *intra*-procedural alias information and *no consideration* of indirect function calls. We refer to these static analyses as *finger traceable analysis* techniques (FiT analysis).

To demonstrate the usefulness of our approach, we designed and implemented FiTx, a framework for generating developer-friendly bug checkers that seamlessly integrate as compiler extensions. FiTx's primary purpose is to scan codebases efficiently and proactively notify developers of potential bug candidates during early development stages, complementing existing state-of-the-art tools like PATA [32], which excel at detecting complex bugs requiring path-sensitive, interprocedural alias analysis in later development phases.

| Tools | | CSA | CppCheck | Coccinelle | Saber | PATA (Data taken from original paper [32]) | FiTx (Our approach) |
|---|---|---|---|---|---|---|---|
| Characteristic | Scope | Unit(Partial) | Unit | Function | Project | Project | Unit |
| | Path Sensitivity | Sensitive(Partial) | Insensitive | - | Sensitive | Sensitive | Insensitive |
| | Alias Analysis | Inter(Partial) | None | - | Inter | Inter | Intra |
| | Field Offset | Partial | Partial | - | Yes | Yes | Yes |
| | Indirect Calls | None | None | - | Yes | None | None |
| Analysis time per file (sec) | Total | 27hr 1min | 2hr 32min | 4hr 20min | OOM | 33 hour 1min | 2hr 33min |
| | 50%tile | 3.33 | 0.03 | 0.48 | - | N/A | 0.20 |
| | 90%tile | 10.75 | 0.32 | 0.81 | - | N/A | 0.99 |
| | 99%tile | 32.85 | 3.00 | 2.46 | - | N/A | 3.79 |

Table 1: Characteristics of State of the art tools

| Bug Type | Keywords |
|---|---|
| Use Before Initialization | use before, use-before, uninitialized |
| Double Free | double free, double-free |
| Out of Bounds | out-of-bounds, OOB, out of bounds |
| Integer Overflow | integer overflow, integer-overflow |
| Null Pointer Dereference | null pointer, null ptr, nullptr |
| Reference Counting Error | ref, kref, ref count |

Table 2: Sampled bug types and the keywords

| Category | | | # of patches |
|---|---|---|---|
| Sampled patches | | | 105 |
| Target analysis scope (Function + Unit) | | | 72 (21 + 51) |
| Detailed Investigation (72 patches) | Field Offset | None* | 36 |
| | | Compile* | 29 |
| | | Runtime | 7 |
| | Alias | None* | 56 |
| | | Intra* | 15 |
| | | Inter | 1 |
| | Control flow | Direct* | 65 |
| | | Indirect | 7 |
| FiT analysis findable bugs (FiT bugs) | | | 64 |

Table 3: Investigation results. FiT bugs are an intersection of patches with $*$ characteristics.

## 3.1 Observation: Impact of Leveraging FiT analysis in Linux

To demonstrate the effectiveness of the FiT analysis to identify existing bugs, we first conduct a simple check of recent bugs in Linux. We check well-known Linux bug patterns by sampling the bug-fixing patches from version 5.9 (released September 2020) to version 5.11 (released February 2021). We use keyword-based patch collection [9, 43] and sample 105 bug-fixing patches that contain keywords such as *double free*, *out-of-bounds*, or *integer overflow*, which are related to 6 well-known bug patterns: Use Before Initialization (UBI), Double Free (DF), Out of Bounds Access (OoB), Integer Overflow (INT), Null Pointer Dereference (NULL) and Reference Counting Error (REF). The full list of keywords is shown in Table 2.

We check the characteristics of each bug-fixing patch from the perspective of static data flow analysis, and determine the level of analysis required to spot the bug. We check each bug under the following static analysis criteria:

1. The analysis scope required to find the bug. We focus on 3 levels of scopes. *Function*, which only requires analysis of a single function, *Unit*, which requires analysis of a single compilation unit, and *Project*, which requires analysis of the entire software.
2. Field offset calculation required to find the bug. Our survey focuses on two types of offset-based values: *Compile*, whose offset is determined statically, and *Runtime*, whose offset is determined dynamically. If the bug does not involve any field calculation, we indicate it as *None*.
3. Level of alias information required to find the bug. We focus on 3 levels of alias information: *None* which does not require alias information, *Intra* which requires intraprocedural alias information, and *Inter* which requires interprocedural alias information.
4. If the bug can be found by traversing direct control flows. Direct control flow means it does not involve indirect function calls.

Table 3 shows the summary of characteristics for the checked patches. Out of all the 105 sampled patches, 72 of the bugs can be found with the analysis of a single compilation unit. Within these cases, 21 of the bugs require an intraprocedural analysis only. According to the above criteria, we further investigate the bugs that can be found by examining a single compilation unit.

**Field offset.** Out of the 72 patches that are contained within a single compilation unit, our investigation shows that

```
drivers/net/ethernet/mellanox/mlx5/core/en_fs.c
1  int create_ttc_table(struct ttc_table *ttc) {
2    err = create_ttc_table_groups(ttc);
3    if (err)
4      destroy_flow_table(&ttc->ft);
5  }
6  int create_ttc_table_groups(struct ttc_table *ttc) {
7    struct flow_table *ft = &ttc->ft;
8    ft->g = kcalloc(...);
9    if (err) {
10     kfree(ft->g);
11 +   ft->g = NULL;
12     return -ENOMEM;
13   }
14 }
15 void destroy_flow_table(struct flow_table *ft) {
16   kfree(ft->g);
17 }
```

(a) Double free bug in driver (7a6eb072a954)

```
net/decnet/dn_route.c
1  int dn_route_output_slow(...) {
2    struct net_device *dev_out = dev_get_by_index(..);
3    ...
4    dev_hold(dev_out); // Inc refcount
5    fld.saddr = ...;
6    if (!fld.daddr) {
7  +   dev_put(dev_out); // Dec recount.
8      return err;
9    }
10 }
```

(b) Refcount bug in net (3f96d6449768)

Figure 2: Motivating examples of FiT analysis findable bug in Linux

only 7 of the bugs require dynamic calculation of field offsets. The remaining 65 patches are either struct-related or access to the array element with a constant index.

**Alias Information.** Our examination reveals that only one patch requires interprocedural alias analysis. The remaining patches can be found with either intraprocedural alias information (15 patches), or no alias information (56 bugs).

**Control Flow.** Our examination reveals that only 7 of the patches involve indirect function calls. The remaining 65 bugs can be found by traversing direct control flows.

Out of the sampled 105 patches, our results show that 64 of the bugs can be found with FiT analyses we leverage. This suggests that leveraging only such elementary analysis methods can still have a positive impact on improving codebase health on finding FiT analysis findable bug (FiT bugs) in Linux.

### 3.1.1 Motivating Example

Figure 2 shows two simplified examples of FiT bugs. Figure 2a is a double free bug in drivers code (7a6eb072a954), which was found manually and took over 1200 days to be fixed [23]. In line 2, function `create_ttc_table_groups` is called to initialize `ttc`. It aliases `ttc->ft` to `ft` in line 7, and allocates `ft->g` in line 8. If the function encounters an error (line 9), it frees `ft->g` and returns an error code. However, when the caller receives an error (line 3), it calls

| Method | Tools | Total |
|---|---|---|
| Static | Compiler | 8 |
|  | Coverity | 3 |
|  | Clang Static Analyzer | 1 |
| Dynamic | Syzkaller | 11 |
|  | Abaci fuzz | 1 |
| Not specified (Prob. Manual) |  | 40 |
| Total |  | 64 |

Table 4: Method used to find FiT bugs

`destroy_flow_table` and frees `ft->g` again (line 16), resulting in double free. The bug is fixed by storing NULL to the freed field in line 11. This bug can be found with an analysis of a single source file, as all of the functions are in the same source file (lines 1, 6, 15). All the function calls are direct (lines 2, 4). In addition, it requires only intraprocedural alias information in line 7, and the field accesses are all struct fields (lines 4, 8, 10, 16).

Figure 2b is a reference counting bug found in net code (3f96d6449768) [51]. This bug took over 3600 days (more than 9 years) to be fixed and found manually by the developers. In line 4, function `dev_hold` is called to increment the internal reference counter in `dev_out`, which is inline expanded to the call to `this_cpu_inc` on a field of `dev_out`. When the function encounters an error (line 6), the function returns the error code to the caller, without decrementing the reference counter, leading to refcount leak. This bug is fixed by adding a call to `dev_put` (line 7) in the error path. This is another example of FiT bugs. It manifests within a single function (`dn_route_output_slow`), involves only struct fields within the two refcount-related functions, and does not require any alias information.

### 3.1.2 State-of-the-art-tools and FiT bugs

Many state-of-the-art tools target FiT bugs because it can be found with a combination of elementary analyses. However, as our survey results in Section 3.1 suggest, there are still many FiT bugs left in Linux.

To understand to what extent bug detection tools are used, we further examine the sampled bug-fixing patches. Bugs found with bug detection tools are associated with some level of credit. For instance, patches generated by Coccinelle [36] have a line with the "Generated by" tag. We categorize the detection methods into three types: static method (e.g. compiler warnings, static analyzer), dynamic method (e.g. syzkaller [4], sanitizer), and not specified (probably manually found by developers).

Table 4 shows the result. Out of the 64 bugs categorized to FiT bugs, 12 of the bugs were found using static analysis methods (8 with compiler warnings, three with Coverity [2], one with Clang Static Analyzer [6]). On the other hand, 12 of the bugs were found using dynamic analysis methods, specifically fuzzers (11 with Syzkaller, one with Abaci Fuzz [55]).

| Rule Name | Hook Inst. | Operand | Constraint |
|---|---|---|---|
| Fun Arg | `call` | arg | arg num |
| Fun Call | `call` | return val | - |
| Store ANY | `store` | storee | - |
| Store NULL | `store` | storee | store null |
| Store NON | `store` | storee | store non null |
| Store Const | `store` | storee | store const value |
| Use | `load` | loadee | - |

Table 5: Example of Transition Rules

The remaining 40 bugs did not mention any detection methods, suggesting a reliance on alternative approaches including manual code inspection. This highlights a need for more developer-friendly tools that seamlessly integrate into their daily workflows.

## 4 FiTx: A Proof of Concept

FiTx is a framework that generates daily-development-friendly bug checkers. To allow customizing to find project- or module-specific bugs, our framework is customizable and allows the developers to specify the bug they want to look for with typestate properties. Our framework generates a compiler plugin that checks the specified bug and can be used alongside the compiler.

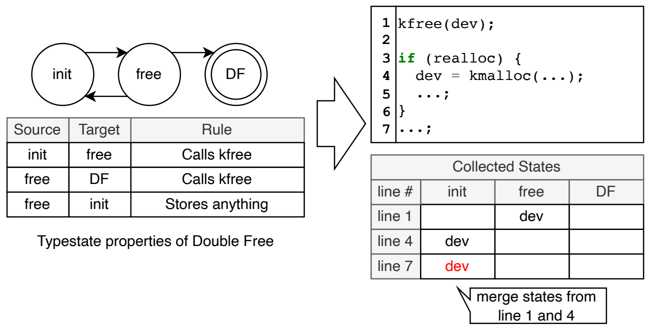
Our analysis is based on the typestate property analysis [20, 22, 26, 32, 41, 52, 56], which takes a definition of each bug expressed in the form of typestate property (finite state machine (FSM)), and traverses the flow graph to check for buggy states. Our analysis considers the buggy state as the accepting state of the FSM property.

With a given typestate property, our extension traverses the Control Flow Graph (CFG). It conducts a path-insensitive, interprocedural, and field-sensitive CFG analysis. To efficiently conduct interprocedural analysis, we conduct a bottom-up summary-based analysis. We also conduct basic intraprocedural and path-insensitive points-to analysis. As shown in Section 3, these analysis techniques can find many bugs in Linux while achieving a short analysis time.

To enhance accuracy within the summary-based analysis, we introduce *return code-aware state propagation*. It summarizes function states for each distinct return code, enabling precise propagation of anticipated states within caller functions by leveraging return code evaluation. This tracking of state transitions within and across functions contributes to a more comprehensive and accurate analysis without sacrificing the analysis time.

### 4.1 Typestate Property Specification

Our framework takes the specification of each bug in the form of typestate property, which is a finite state machine associated with each variable. Like the traditional typestate analysis, our analysis associates the states with each variable,


init
free
DF
Typestate properties of Double Free
Collected States
merge states from line 1 and 4

| Source | Target | Rule |
|---|---|---|
| init | free | Calls kfree |
| free | DF | Calls kfree |
| free | init | Stores anything |

```
1 kfree(dev);
2
3 if (realloc) {
4   dev = kmalloc(...);
5   ...;
6 }
7 ...;
```

| line # | init | free | DF |
|---|---|---|---|
| line 1 | | dev | |
| line 4 | dev | | |
| line 7 | dev | | |

Figure 3: Basic flow of collecting type states

and transitions the states according to the transition rules. The developers can specify the transition rules to define project- or module-specific bugs.

Table 5 shows the list of example transition rules. A transition rule consists of three components: hook instruction, operand constraint, and target operand. *Hook instruction*, which is specified in the form of LLVM IR [31], triggers the transition. *Operand Constraint* is a set of constraints for the transition to be triggered. Our framework checks if all the constraints are met, and considers that the transition is triggered if and only if they are satisfied. *Target Operand* is the operand of the hook instruction which the state is associated with.

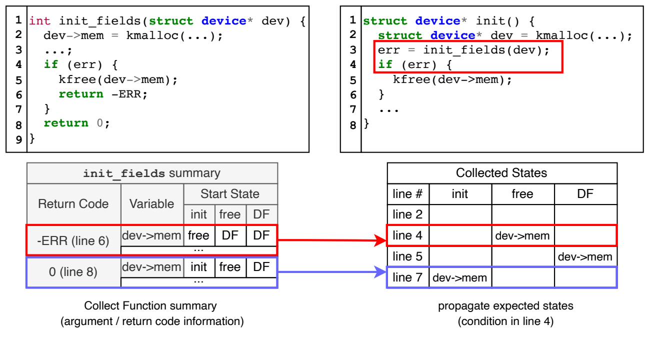
### 4.2 CFG-based Typestate Analysis

Using the defined typestate properties, our analysis conducts a field-sensitive, path-insensitive, and interprocedural analysis to collect the states of each variable. Our analysis traverses each basic block in the CFG and determines if the variable may satisfy the buggy state.

For each basic block, our analysis collects the state of each variable from the predecessor and determines the current state. If all the predecessors have the same state, we propagate the same state as the current state. Otherwise, there is a need to determine the state to be propagated to the successor.

To determine the state to propagate, we define the priority of each state and propagate the highest prioritized state to the successor. By default, our analysis defines the priority of a state using the distance to the bug state. If the distance is large (i.e. the state is further away from the bug state), we consider that the state has greater priority. This is a conservative strategy to suppress false positives. This strategy can be customized by the developers when defining the typestates.

After propagating the states, our analysis traverses each instruction in the basic block and checks if it triggers any transitions. When detecting a transition rule, we check if the current state of the target variable matches the source state of the transition, and update the state according to the provided rule. Any variable without a state is considered as the `init`



```
1 int init_fields(struct device* dev) {
2   dev->mem = kmalloc(...);
3   ...;
4   if (err) {
5     kfree(dev->mem);
6     return -ERR;
7   }
8   return 0;
9 }
```

```
1 struct device* init() {
2   struct device* dev = kmalloc(...);
3   err = init_fields(dev);
4   if (err) {
5     kfree(dev->mem);
6   }
7   ...
8 }
```

| init_fields summary | | | | |
|---|---|---|---|---|
| Return Code | Variable | Start State | | |
| | | init | free | DF |
| -ERR (line 6) | dev->mem | free | DF | DF |
| | ... | | | |
| 0 (line 8) | dev->mem | init | free | DF |
| | ... | | | |

| Collected States | | | |
|---|---|---|---|
| line # | init | free | DF |
| line 2 | | | |
| line 4 | | dev->mem | |
| line 5 | | | dev->mem |
| line 7 | dev->mem | | |

Collect Function summary
(argument / return code information)
propagate expected states
(condition in line 4)

Figure 4: Example of summary-based interprocedural analysis with return code aware propagation

state.

Figure 3 shows a simple example of the flow of our analysis. This example defines a simplified typestate property for double free, and looks for the states in the code fragment. In line 1, `kfree` is called with argument `dev`. This triggers two transitions: `init` to `free` and `free` to `DF`. Since `dev` is in `init` state, the state of the variable will be transitioned to `free`. This state is propagated directly to the basic block in line 3, since the block does not have any other predecessors. Then, in line 4, the state of the `dev` is transitioned again to `init`, since the return value of `malloc` is stored and the source state is `free`. The block in line 7 has 2 predecessors: line 1 and line 3, in which both blocks have different states for `dev`. Since the distance from `init` to the buggy state (`DF`) is longer, the `init` state is propagated to the block.

## 4.3 Summary-based interprocedural analysis

Our analysis checks for the possible buggy states interprocedurally. To efficiently collect the states, we conduct a bottom-up interprocedural analysis using function summaries. When analyzing each function, we collect the states for each argument and the return value and summarize the states at the end of the function. When the function is called, we return the summarized states to the caller without re-analyzing the function.

Since the initial state of the argument cannot be concretely determined without the caller function, our analysis assumes that the argument may start from any state, and record the transitions for all the states in the typestate property. When a function is called, the destination states corresponding to the arguments' states are returned to the caller. If the called function is defined outside of the compilation unit, we consider the state of the argument as unpredictable, and omit from being traced afterward.

**Return Code Aware State Propagation** Some functions have multiple return points, and each point may have a different set of states. The caller, on the other hand, expects the callee to return at a certain point. Neglecting these contexts results in imprecise analysis and leads to many false

| hook name | description |
|---|---|
| IMMEDIATE | immediately after transition (default) |
| VAR_END | end of variable lifetime |
| BLOCK_END | end of each basic block |
| FUNC_END | end of each function |
| MOD_END | end of the analysis |

Table 6: List of bug-checking hooks

positives. For instance, the caller may have an error handling path that expects the callee to end in an error state.

To overcome these cases, we take into account the return code of each function when creating the summary, and attempt to propagate the states which satisfy the caller's context. Our analysis leans on the fact that constant integer return values are often used to express a returning state of each function, known as return code. In the Linux coding convention, a negative constant integer expresses an error code, while a non-negative, non-constant number expresses a success code [35].

Our analysis collects the possible return codes of each function. Constant return values are collected since non-constant values are known to express success codes. If the returned variable contains any constant value (checked by traversing the use-def chain), the states of the basic block returning that constant are collected and are considered the return state for that code. If multiple blocks return the same value, we merge the states and consider the merged states the return states.

For each function call, our analysis determines if the caller expects the callee to return a certain return code. This is achieved by checking if the return value is used in a branch condition. If it is, the states of the return code that satisfies the condition are propagated to the caller. If multiple return codes satisfy the condition, we merge the states of those return codes and propagate back the merged states. If the return value is not checked, we assume that the caller expects the success state from the callee.

Figure 4 shows an example of summary-based interprocedural analysis with return code consideration. Function `init_fields` has one argument, `dev`, whose field `mem` is freed in line 5. We keep track of the transitions for all the states in the typestate; `init` transitions to `free`, `free` transitions to `DF`, while `DF` makes no transition. Then, we check the return statement in the function to determine the states to summarize and collect the error codes. The return statements are placed in line 6 (`-ERR`) and line 8(`0`). Both of the return codes and the corresponding states are included in the function summary. When `init_fields` is called in line 3, we check the usage of return value `err`, and pass the expected states. Here, we pass the state of return code `-ERR` to line 5, and the state for `0` to line 7.

## 4.4 Generating Warnings

With the collected states, our analysis generates warnings to variables that reach buggy states. By default, it is checked after each transition whether any variable is in a buggy state or not. It is necessary to check for the states in other situations. Our analysis provides multiple checkpoints as listed in Table 6: at the end of basic block (`BLOCK_END`), function (`FUNC_END`), or variable lifetime (`VAR_END`). `MOD_END` indicates the states should be checked at the end of the analysis, which is used for functions that have no callers inside the same compilation unit. The developers can specify which checkpoint to be used to check for the bug states.

## 5 Implementation

Our framework is implemented using the LLVM compiler framework [31] with Clang C compiler [1]. As mentioned in Section 4, our framework takes typestate definitions of each bug. A C++ interface is provided to define the states and the transition rules. Our framework generates a Clang plugin from the typestate definitions. For efficiency, our framework takes multiple typestate definitions and bundles them into a single plugin. Developers can run the plugin by building their source files with the plugin-enabled Clang compiler. Object files are generated by the Clang compiler, and the plugin generates the warnings. The developers can specify the embedded compiler with `CC` and `KCFLAGS` flags when running the Linux `Makefile` without modifying anything.

We have implemented checkers for 6 well-known bug patterns: Double Free (DF), Double Lock / Unlock (DL / DUL), Memory Leak (ML), Use After Free (UAF) and Ref Count Error (Ref). Each plugin consists of less than 5 states and transitions, and requires less than 50 Lines of C++ code. Coccinelle and CSA require 134 and 3,428 LoC respectively to implement a use-after-free checker.

**Supporting domain-specific semantics** Our framework supports supplying the plugin with additional domain-specific semantics via the C++ interface. Developers can use the interface to embed semantic assertions or conventions that FiTx can take into account. This allows FiTx to filter out false positives that are obvious to developers with domain-specific knowledge, but still require semantic information. In our plugins, we embed generically applicable rules found in a wide range of components (e.g. functions which contain `put` in their name are typically reference count-decrementing functions).

## 6 Evaluation

Using the implemented checkers, we analyze the Linux Kernel v5.15, the latest version at the time of the writing. We first determine if FiTx can find bugs in Linux, and introduce the bug found by our analysis. We then determine the analysis time required per file, as well as the number of warnings generated per file. We also compare our method with two well-known state-of-the-art tools used in daily development, CSA and CppCheck, and determine whether they can also find the bugs as well as the analysis time required per each source file.

| | |
|---|---|
| OS | Ubuntu 20.04 |
| CPU | 16 Core Intel Xeon CPU E5-2620 |
| RAM | 96 GB (limited to 32 GB) |
| LLVM | 10.0.1 |
| Target Kernel | v5.15 |
| Config | `allyesconfig` |

Table 7: Experimental Setup

| Bug Type | Warnings | True Positives |
|---|---|---|
| DF | 38 | 21 |
| DL | 15 | 7 |
| DUL | 11 | 5 |
| ML | 16 | 3 |
| UAF | 29 | 9 |
| Ref | 4 | 2 |
| Total | 113 | 47 |

Table 8: Number of generated warnings

Table 7 shows the setup for our analysis. To emulate the environment of daily development, we limit usable memory to 32GB. For kernel configuration, we use `allyesconfig` and manually turn off the components that cannot be compiled using Clang. In our environment, we can compile a total of 20,634 files. We run the build with the debug option (`-g`). Because the Linux kernel cannot be compiled with the optimization disabled, we add a compiler pass to preserve a non-optimized LLVM-IR.

### 6.1 Analysis Results

#### 6.1.1 Number of bugs found

To demonstrate that FiTx can find real bugs, we first build the Linux with FiTx enabled, and check the generated warnings to determine the true positives.

Table 8 shows the number of warnings generated by the checkers per bug type. In total, FiTx generated 113 warnings. Our manual inspection revealed that 47 of the warnings pointed to actual bugs. Out of the found bugs, double free had the highest number of true positives with 21 cases, followed by use after free with 9 cases, double lock with 7 cases, double unlock with 5 cases, memory leak with 3 cases, and reference counting bug with 2 cases. We further created bug fix patches for 16 of the found bugs which could be fixed easily, and reported to the developers. As of the time of the writing, 13 bugs have been confirmed and merged [42,44–48]. The remaining patches are still waiting for a response from the developers.

Figure 5 shows a simplified example of a bug found in a GPU driver code. This bug was initially found as a double free bug but later, an investigation revealed that it triggers multiple types of bugs including memory leak, uninitialized

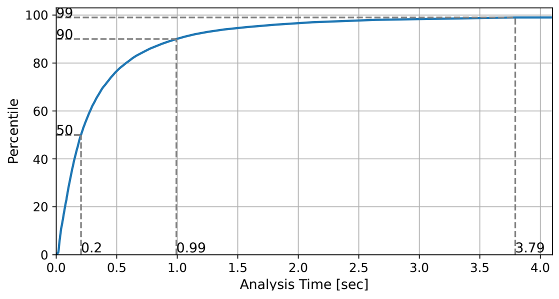
```
drivers/gpu/drm/amd/pm/legacy-dpm/si_dpm.c
 1 int si_dpm_sw_init(void *handle) {
 2   struct amdgpu_device *adev = (...)handle;
 3   ret = si_dpm_init(adev);
 4   if (ret)
 5     si_dpm_fini(adev);
 6   ...
 7 }

 8 int si_dpm_init(struct amdgpu_device *adev) {
 9   ret = si_parse_power_table(adev);
10   if (ret)
11     return ret;
12   ...
13 }

14 int si_parse_power_table(struct amdgpu_device *adev) {
15   adev->pm.dpm.ps = kcalloc(num_entries, ...);
16   for (int i = 0; i < num_entries; i++) {
17     ps = kzalloc(...);
18     if (ps == NULL) {
19       kfree(aev->pm.dpm.ps);
20       return -ENOMEM;
21     }
22     adev->pm.dpm.ps[i].ps_priv = ps;
23   }
24   adev->pm.dpm.num_ps = num_entries;
25   return 0;
26 }

27 void si_dpm_fini(struct amdgpu_device *adev) {
28   if (adev->pm.dpm.ps)
29     for (int i = 0; i < adev->pm.dpm.num_ps; i++)
30       kfree(adev->pm.dpm.ps[i].ps_priv);
31   kfree(adev->pm.dpm.ps);
32 }
```

Figure 5: Bug found in `si_dpm` code. It causes double free (line 19, 31), memory leak (line 19), uninitialized variable access (line 29), null pointer dereference, array index out of bounds, and use after free (line 30)

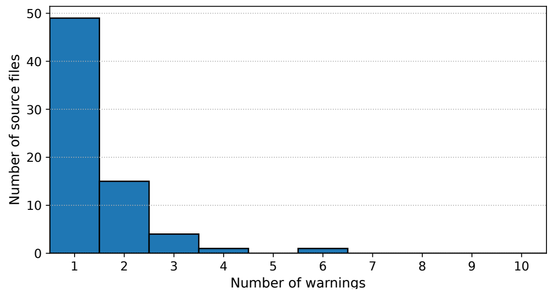
variable access, null pointer dereference, array index out of bounds access, and use after free.

In function `si_dpm_sw_init()`, `si_dpm_init()` is called to initialize various fields of variable `adev` (line 3). This function calls `si_parse_power_table()` to allocate array `adev->pm.dpm.ps` (line 15) and its elements (lines 16 ~23). If the allocation of an element fails (line 18), it frees the array and returns an error code (line 19, 20). However, the array is freed again in `si_dpm_init` (line 31) without being NULLed, which is called when the initialization fails(line 5), resulting in double free of the array.

In addition, since the array is freed without freeing the array elements, these elements are leaked, resulting in a *memory leak* (line 19). Moreover, `si_dpm_fini` attempts to free the array elements in the loops in lines 29 and 30 using `adev->pm.dpm.num_ps` in its condition. This struct field is only initialized if the allocation successfully finishes (line 24), resulting in the *uninitialized variable access* in line 29, also triggering *array index out-of-bounds access* or *NULL pointer dereference* in line 30. On top of that, line 30 accesses an already freed array, causing *use after free* of the array. This bug existed since 2016 and was fixed by us.



(a) CDF of analysis time per file



(b) Histogram of number of warnings generated per file

Figure 6: Analysis results per file

### 6.1.2 Analysis result per source file

We examine the analysis time required for each compilation unit to determine the wait time required by the developers. Figure 6a shows the CDF of analysis time. FiTx only required 0.20 seconds for 50% of the source file, and 0.99 seconds for 90% of the source file. When comparing the longer analysis time (99%tile value), FiTx finished in 3.79 seconds, which is around 8.5 times shorter analysis time than CSA (Table 1).

We also examine the number of warnings generated per file to determine how many warnings developers are required to check. Figure 6b shows the histogram of the number of warnings generated per file. 99.9% of the source files only generated at most three warnings per each source file. For two source files, FiTx generates 4+ warnings. This is due to a bug candidate in a function being propagated to many callers. These warnings are easily spottable since they mention the involvement of the same function.

## 6.2 Comparison with Other Methods

### 6.2.1 Comparison with Framework Variants

We first compare our approach with variants of our framework. These variants differ from our proposed approach by leveraging more complex/simple approaches. In this work, we compare with three variants: (a) FiTx-PS: complex variant which leverages path-sensitive approach for higher precision (originally path-insensitive), (b) FiTx-NR: simple variant which does not conduct return-code considered state prop-

| Variant | Proposal | Variants | | |
|---|---|---|---|---|
| | | PS | NR | PSNR |
| Target (Files) | 20,634 | | | |
| Timed out (Files) | 0 | 5,068 | 0 | 5,014 |
| Compiled (Files) | 20,634 | 15,566 | 20,634 | 15,620 |
| Total Time | 2hr 33min | 13hr 38min | 2hr 9min | 13hr 20min |
| Total Warnings | 113 | 89 | 82 | 86 |
| True Positives | 47 | 11 | 27 | 10 |

Table 9: Analysis of Linux Kernel for framework variants. Variant description: *NR*: does not conduct return-code aware state propagation, *PS*: conducts path-sensitive analysis, *PSNR*: conducts path-sensitive analysis and does not conduct return-code state propagation.

agation (originally conducted), and (c) FiTx-PSNR: cross-moderate variant which leverages path-sensitive approach without return-code considered state propagation. We first conduct the same analysis of the Linux Kernel as the proposed approach to determine if the source files can be analyzed within the time limit and if the bugs can be found. We then determine the required analysis time for each analyzed source file to determine the additional compilation time required for each variant.

We first analyze the entire Linux kernel using the same method as Section 6.1. We limit the usable memory to 32 GB and set the timeout of each compilation to one minute to emulate the daily development of developers. Table 9 shows the results of the analysis. Overall our approach was able to find the most bugs and analyze the entire Linux kernel within the time limit. Variants that invoke path-sensitive approach (PS and PSNR) did not finish the analysis for more than 5,000 source files (5,068 and 5,014 respectively) because of the time limit, taking more than 13 hours of analysis time for both cases. The path-insensitive approaches (Proposal and NR), on the other hand, were able to finish analyzing all the source codes within the time limit (2hr 33 min and 2hr 9min respectively). Variants which does not invoke return-code considered state propagation were able to finish the analysis slightly faster than the variants which leverage the same path sensitivity (around 20 minutes faster). However, the non-considered variants could not find some of the bugs which the considered variants could find (20 and 37 bugs were not found respectively).

We further determine the required analysis time of each variant and whether each variant can find the bugs by analyzing the source files which include developer-confirmed bugs. For the analysis of each variant, we set the timeout of each analysis to 12 hours. Columns indicated *FiTx variants* in Table 10 show the result of this analysis. Like the analysis of the entire Linux Kernel, our approach was able to find the most number of bugs in each source file within the time limit. Variants that use path-sensitive approaches (PS and PSNR) required significantly long analysis time for 4 source files and required more than 4 hours to finish analysis, with 2 source files timing out (required more than 12 hours). It required 1.5 ~103,714 times more analysis time than the proposal. Variants that did not conduct return-code considered state propagation (NR and PSNR) could not find some of the bugs (3 and 7 bugs respectively). Although NR finished analyzing each source file faster than the proposal (average 1.2 times faster), it did not find the bugs that the proposal found.

### 6.2.2 Comparison with Existing Tools

We compare our approach with two of the well-used tools in daily development: Clang Static Analyzer (CSA, v10.0.1) and CppCheck (v1.90). We first determine if the bugs can be found with the tools. We analyze the previously found bugs (NULL pointer dereference and double free) from Section 3.1 and the developer-confirmed true positives from our analysis. We then determine the required analysis time for each analyzed source file to determine the additional compilation time. We run each analyzer with the default configuration. For CSA, we additionally conduct the analysis using two different configurations: Unix-specific config (`clang-analyzer-unix.*`) which limits the target bugs to Unix-specific bugs, and Unix-specific memory bug config (`clang-analyzer-unix.Malloc`) which further limits the target bugs to Unix-specific memory bugs.

**Number of Detected Bugs** We first conduct an analysis of the known bugs found in Section 3.1. Due to a large number of patches, we specifically focus on the patches which fix NULL pointer dereferences and double frees. In total, we analyze 8 patches. We created a customized extension for FiTx. For CSA we only run the analysis with the default configuration because the NULL checker does not exist in the remaining two Unix-specific configurations. Table 11 shows the results. Out of the 8 patches, FiTx can find 7 bugs correctly. The remaining bug (862aecbd9569) could not be found due to the lack of configuration in our environment. CSA can find 1 bug correctly, and CppCheck does not find any of the bugs.

We further check the analysis results for the source files which contain developer-confirmed true positives. All of the bugs are either double free or use-after-free which is the target of both CSA and CppCheck. Table 10 shows the analysis results per source file. Unfortunately, both of the tools did not find any of the bugs.

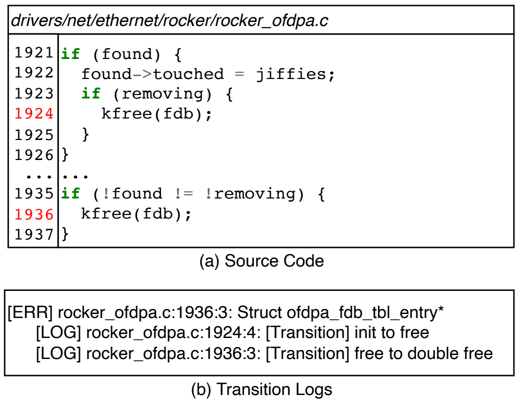
The reason for the false negatives of state-of-the-art tools is their analysis limitations. The majority of the false negatives in CSA are caused by the strong limitation in the interprocedural analysis. As mentioned in Section 3.1.2, CSA only analyzes 1 function call depth to keep the analysis time short. However, many bugs involve multiple depths of function calls. For instance, Figure 5 involves at least 3 function call depths. This accounts for 5 cases of known bugs and all of the cases of confirmed bugs. The remaining 2 cases of known bugs are caused by the lack of domain-specific knowledge where it did not consider the module-specific allocation function. The

| File Name | LoC | TP | Criteria | FiTx | Clang Static Analyzer (CSA) | | | CppCheck | FiTx Variants | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Default | Unix | Unix.Malloc | | NR | PS | PSNR |
| drivers/platform/ chrome/chromeos_laptop.c | 958 | 2 | Detected | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| | | | Time (sec) | 0.14 | 5.85 | 4.19 | 3.82 | 0.03 | 0.11 | 0.21 | 0.13 |
| drivers/media/dvb-core/dvbdev.c | 1,084 | 1 | Detected | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| | | | Time (sec) | 0.14 | 8.34 | 5.56 | 5.53 | 0.08 | 0.34 | 4hr 2min | 4hr |
| kernel/trace/trace_events_hist.c | 6,113 | 6 | Detected | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ |
| | | | Time (sec) | 1.69 | 43.01 | 32.56 | 32.52 | 7.28 | 1.39 | 7hr 47min | 7hr 46 min |
| drivers/gpu/drm/amd/pm/ powerplay/si_dpm.c | 7,127 | 2 | Detected | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | - | - |
| | | | Time (sec) | 2.18 | 24.59 | 16.62 | 16.53 | 0.89 | 1.70 | T.O. | T.O. |
| drivers/scsi/qla2xxx/qla_os.c | 8,216 | 2 | Detected | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | - | - |
| | | | Time (sec) | 2.80 | 41.57 | 30.26 | 29.77 | 1.50 | 2.53 | T.O. | T.O. |

Table 10: # of warnings and analysis time per source file which contains confirmed bugs. Clang Static Analyzer is measured with three configurations: *Default*: default configuration used in Linux, *Unix*: Unix specific bug only configurations (`clang-analyzer-unix.*`), *Unix.Malloc*: Unix specific memory bug only configurations (`clang-analyzer-unix.Malloc`).

| hash | dir | Originally found by | FiTx | CSA | CppCheck |
|---|---|---|---|---|---|
| 73644143b31c | drivers | manual | ✓ | ✗ | ✗ |
| b097efba9580 | drivers | coverity | ✓ | ✗ | ✗ |
| 733c15bd3a94 | drivers | coverity | ✓ | ✓ | ✗ |
| 862aecbd9569 | drivers | manual | - | - | ✗ |
| 13384f6125ad | mm | syzkaller | ✓ | ✗ | ✗ |
| 292bff9480c8 | drivers | coverity | ✓ | ✗ | ✗ |
| 7a6eb072a954 | drivers | manual | ✓ | ✗ | ✗ |
| dad1b1242fd5 | fs | Abaci fuzz | ✓ | ✗ | ✗ |

Table 11: Analysis results of NULL pointer dereference and double free bugs in Table 4. ✓: found. ✗: not found. -: the patch could not be analyzed in our environment.


drivers/net/ethernet/rocker/rocker_ofdpa.c

```
1921 if (found) {
1922   found->touched = jiffies;
1923   if (removing) {
1924     kfree(fdb);
1925   }
1926 }
 ... ...
1935 if (!found != !removing) {
1936   kfree(fdb);
1937 }
```

(a) Source Code

```
[ERR] rocker_ofdpa.c:1936:3: Struct ofdpa_fdb_tbl_entry*
    [LOG] rocker_ofdpa.c:1924:4: [Transition] init to free
    [LOG] rocker_ofdpa.c:1936:3: [Transition] free to double free
```

(b) Transition Logs

Figure 7: Example of false positive produced by our tool

false negatives of CppCheck are caused by the fact that it does not conduct any complex analysis. For instance, CppCheck does not conduct inter-procedural analysis of pointer value in certain situations. This leads to many bugs being overlooked.

**Analysis Time** We compare the analysis time of each source file for the analysis of found bugs. Table 10 shows the results. Compared to CSA, FiTx finished the analysis faster. CSA conducts path-sensitive analysis and the analysis time grows exponentially when the number of paths grows, typically dominated by the line of code. It required 5.85 seconds for the source file with less than 1,000 LoC, which is around 40 times slower than FiTx. CSA required over 40 seconds (14.9x slower than FiTx) for the source file with 8,200+ LoC (qla_os.c), and even when limiting the target with config, it still required 29.77 seconds (10x slower than FiTx).

CppCheck finished the analysis faster than FiTx. It only required 1.50 seconds in qla_os.c, while FiTx required 2.80 seconds, which is 1.86x faster. This is because CppCheck does not conduct complex analysis such as tracing the pointer values interprocedurally. FiTx finished the analysis faster (2.18 sec) than CppCheck (7.28 sec) for one case (trace_events_ hist.c). This is because CppCheck checks all the macro configuration patterns to find the bugs, checking 20 patterns in this case. Although it generally incurs less analysis time, it misses all the analyzed bugs.

| Bug Type | Path Sensitivity | Context Sensitivity | Semantical |
|---|---|---|---|
| DF | 4 | 7 | 6 |
| DL | 1 | 4 | 3 |
| DUL | 0 | 1 | 5 |
| ML | 2 | 3 | 8 |
| UAF | 13 | 4 | 3 |
| Ref | 0 | 1 | 1 |
| Total | 20 | 20 | 26 |

Table 12: False positive causes of FiTx

## 7 Discussions

**False Positives** Static bug detection tools often face the challenge of false positive filtering, posing a significant hurdle to developer adoption. The manual effort involved in identifying and discarding false positives can hinder tool integration. While many tools address them by lowering their rates, FiTx addresses this by only generating a small number of warnings in the first place, alleviating the effort required from developers. It generates 113 warnings, while CSA and CppCheck generated 132,196 and 1,538 respectively. FiTx generated 66 false positives. As mentioned in Section 6.1.2, the developers will only need to check at most three warnings per source file for 99.9% of the cases. Overall it only took less than two minutes on average for an inexperienced Ph.D. student to check each warning. In addition, the false positive rate (58.4%) is smaller compared to the reported rates of CSA and CppCheck (83.0% and 84.3%) [32].

FiTx also generates transition logs to help the developers debug more efficiently and determine if the warning is a false positive or not. Figure 7 shows a false positive and the transition log generated by FiTx. It detected a possible double free in line 1936 and the first transition occurs in line 1924 (init to free state). The developers can trace the state transitions and spot that this is a false positive since the condition clauses for the first transition (lines 1921, 1923) and the second transition contradicts (line 1935).

The false positives can be categorized into three types: (1) lack of path sensitivity, (2) lack of context sensitivity and (3) lack of semantic information. Table 12 shows the distribution of the causes. The majority of the cases were due to the "text-book level" analysis techniques used in FiTx: lack of path sensitivity and context sensitivity with 20 cases each. The former occurs because we do not conduct path-sensitive analysis (e.g. Figure 7), and the latter is caused by the bottom-up interprocedural analysis.

The lack of semantic information is another cause of false positives with 26 cases. FiTx found a bug candidate on the path which will not occur semantically. For instance, driver code typically interacts with the device, where the paths change between multiple device states. In this case, the path will semantically contradict, but the code path-wise may seem independent. These cases are easily spottable by the developers because they are already familiar with these semantics. In addition, developers can further eliminate these false positives by embedding this information into FiTx to prevent it from surfacing. As mentioned in Section 5, FiTx allows to add project/module-specific rules. Developers typically only work on a specific sub-module of the Linux kernel (e.g. block devices, specific driver code) and have specialized knowledge. Hence, they can add tailored rules for each module, such as how state transitions within each function, or making an allow list of functions that reduce false positives.

**False Negatives** Our analysis also generates many false negatives. Like the false positives, the reason is due to the simpleness of the analysis approach, limiting the bugs that could be uncovered to FiT bugs. To find complicated bugs, there is a need to conduct a set of more complex analyses, or expand the scope to the entire software. For instance, indirect function call analysis should be used to find more bugs within indirect control flows. As mentioned in Section 3, such analyses are the scope of the sophisticated, state-of-the-art bug detectors such as PATA, and are not the scope of FiTx where its focus is short analysis time for daily-development use. By combining the usage of differently characterized tools, we believe that many bugs are exposed to the developers.

## 8 Related Work

**Bugs in Linux kernel** There has been much work that studies the characteristics of the bug in Linux [16, 37]. Chou et al. studied the bugs in versions 1.0 to 2.4.1 using a static analyzer [16]. Palix et al. also investigated the bugs in Linux of later versions from 2.6.0 to 2.6.33 [37]. To illustrate how Linux evolved from previous versions, they investigated the bugs using the same criteria as Chou et al. and showed that components other than drivers also suffer from many bugs.

**Static Analysis Methods** Many research efforts propose to find bugs in the Linux kernel using static analysis [2,3,6–14, 17, 19, 24, 25, 30, 33, 34, 38, 43, 49, 50, 53, 54, 58, 59]. Since the Linux kernel consists of millions of lines of code, these methods are required to be scalable without losing precision. These research efforts focus on specific bug patterns (often related to the Linux semantics), and elaborate conventional methods so as to be applied to Linux. For instance, DCUAF [9] targets use-after-free bugs, and DSAC [13] targets inappropriate sleep in non-blocking contexts.

Coccinelle [36] is a commonly used tool in Linux. It allows the developers to describe "semantic patches" to help transform the code when modifying an already existing API. Coccinelle can check for violations of code patterns that lead to bugs. However, it does not find interprocedural dataflow bugs [36, 40]. PATA [32] is a bug detection framework tailored to the operating system. Their work achieves scalable analysis that conducts path-sensitive points-to analysis while finishing the analysis in around 30 hours for Linux.

There are many bug detection tools publicly available. Saber [57] targets generic memory leaks independent of the project semantics. Unfortunately, it is not scalable since it does not assume software systems of millions of lines of code. Clang Static Analyzer [6] faces the hurdle of long analysis time, and CppCheck [19] faces an overwhelming number of warnings.

The proposed FiTx complements the use of these tools. By finding FiT bugs before the use of these sophisticated tools, the number of warnings the tools raise will decrease, and the effort developers need to make to look through the warnings will also decrease. This allows them to concentrate on more complex bugs.

**Dynamic Analysis Methods** Some tools leverage dynamic analysis to find bugs [4, 5, 15, 18, 21, 27–29, 39]. They attempt to find bugs on demand by directly monitoring runtime behavior and checking if the Linux Kernel is behaving correctly. Sanitizers such as Kmemleak [29] or KASAN [5] are built-in to the Linux kernel. It keeps track of the valid memory by shadowing the memory operations and tracing accesses.

Fuzzers are widely used to find bugs in Linux. Syzkaller [4] conducts coverage-guided fuzzing to automatically generate a sequence of system calls and attempts to execute target code paths to uncover as many bugs as possible. While the fuzzers can find complicated bugs with low false positive rates, an extensive amount of analysis time is required and the seeds of the test cases must be carefully designed to achieve high coverage.

## 9 Conclusion

This paper tackles the persistent trade-off between analysis time and bug detection capabilities of static bug detection tools, aiming to promote their adoption in daily development workflows. In daily development, developers prioritize tools offering short analysis time and high bug-finding efficiency. Existing tools often prioritize one aspect at the expense of the other. We propose a combination of less-computationally complex analyses which can find many bugs while achieving a short analysis time, and designed and implemented FiTx. Despite the text-book level analysis, our checker finds 47 bugs (13 bugs confirmed by developers) in Linux v5.15 with 0.99 seconds of analysis time for 90% of the source files.

## Artifact Availability

Our prototype implementation of FiTx is available at https://github.com/sslab-keio/FiTx.

## Acknowledgments

We thank the anonymous reviews for their valuable and insightful feedback.

## References

[1] Clang Compiler. http://clang.llvm.org/.

[2] Coverity. https://scan.coverity.com.

[3] Linux Driver Verification. http://linuxtesting.org/ldv.

[4] Syzkaller: an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[5] The Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[6] Clang Static Analyzer. https://clang-analyzer.llvm.org/, 2022.

[7] Facebook Infer: a tool to detect bugs in Java and C/C++/Objective-C code. https://fbinfer.com/, 2022.

[8] Smatch: a static bug-finding tool for C. http://smatch.sourceforge.net/, 2022.

[9] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency Use-After-Free bugs in linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 255–268, 2019.

[10] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Effective detection of sleep-in-atomic-context bugs in the linux kernel. *ACM Trans. Comput. Syst.*, 36(4):1–30, April 2020.

[11] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Effective detection of sleep-in-atomic-context bugs in the linux kernel. *ACM Trans. Comput. Syst.*, 36(4), apr 2020.

[12] Jia-Ju Bai, Tuo Li, and Shi-Min Hu. DLOS: Effective static detection of deadlocks in OS kernels. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 367–382, 2022.

[13] Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, and Shi-Min Hu. Dsac: Effective static analysis of Sleep-in-Atomic-Context bugs in kernel modules. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 587–600, 2018.

[14] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. In *Formal Methods in Computer Aided Design*, pages 35–42. ieeexplore.ieee.org, October 2010.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[16] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 73–88, New York, NY, USA, 2001. Association for Computing Machinery.

[17] Kai Cong, Fei Xie, and Li Lei. Symbolic execution of virtual devices. In *Proceedings of the 2013 13th International Conference on Quality Software*, QSIC '13, pages 1–10, USA, July 2013. IEEE Computer Society.

[18] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2123–2138, New York, NY, USA, October 2017. Association for Computing Machinery.

[19] CPP Check. cloc. https://cppcheck.sourceforge.io/, 2022.

[20] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, May 2002. Association for Computing Machinery.

[21] P Deligiannis, A F Donaldson, and Z Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177, November 2015.

[22] Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-Sensitive dataflow analysis with iterative refinement. In *Static Analysis*, pages 425–442. Springer Berlin Heidelberg, 2006.

[23] Dinghao Liu. net/mlx5e: Fix two double free cases. https://github.com/torvalds/linux/commit/7a6eb072a9548492ead086f3e820e9aac71c7138, 2021.

[24] Navid Emamdoost, qiushi wu, kangjie lu, and Stephen McCamant. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *The Network and Distributed System Security Symposium (NDSS) 2021*, 02 2021.

[25] G Fan, R Wu, Q Shi, X Xiao, J Zhou, and C Zhang. SMOKE: Scalable Path-Sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82, May 2019.

[26] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, May 2008.

[27] Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. ACHyb: a hybrid analysis approach to detect kernel access control vulnerabilities. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 316–327, New York, NY, USA, August 2021. Association for Computing Machinery.

[28] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Fuzzing error handling code in device drivers based on software fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 128–138, October 2019.

[29] Kernel.org. Kernel Memory Leak Detctor(Kmemleak). https://www.kernel.org/doc/html/v4.17/dev-tools/kmemleak.html, 2019.

[30] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX-ATC'10, page 12, USA, June 2010. USENIX Association.

[31] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[32] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 859–872, New York, NY, USA, 2022. Association for Computing Machinery.

[33] K Lu, A Pakki, and Q Wu. Detecting Missing-Check bugs via semantic-and Context-Aware criticalness and constraints inferences. *USENIX Security Symposium (USENIX Security 19)*, 2019.

[34] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: Finding reference count bugs with inconsistent path pair checking. *SIGARCH Comput. Archit. News*, 44(2):531–544, March 2016.

[35] Paul D. Marinescu and George Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4), December 2011.

[36] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, April 2008.

[37] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

[38] S Saha, J Lozi, G Thomas, J L Lawall, and G Muller. Hector: Detecting Resource-Release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2013.

[39] S Schumilo, C Aschermann, R Gawlik, and others. kAFL:Hardware-Assisted feedback fuzzing for OS kernels. *26th USENIX Security*, 2017.

[40] Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, Lingxiao Jiang, David Lo, Julia Lawall, and Gilles Muller. SPINFER: Inferring semantic patches for the linux kernel. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 235–248. USENIX Association, July 2020.

[41] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[42] Rustam Subkhankulov. platform/chrome: fix double-free in chromeos_laptop_prepare(). https://lore.kernel.org/lkml/20221019083306.044452229@linuxfoundation.org/, 2022.

[43] K Suzuki, T Kubota, and K Kono. Detecting struct member-related memory leaks using error code analysis in linux kernel. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 329–335, October 2020.

[44] Keita Suzuki. drm/amd/pm: fix double free in si_parse_power_table(). https://lore.kernel.org/lkml/20220419103721.4080045-1-keitasuzuki.park@sslab.ics.keio.ac.jp/, 2022.

[45] Keita Suzuki. media: dvb-core: Fix double free in dvb_register_device(). https://www.spinics.net/lists/stable-commits/msg281850.html, 2022.

[46] Keita Suzuki. platform/chrome: chromeos_laptop - fix potential double free. https://lore.kernel.org/lkml/20220314030337.777685-1-keitasuzuki.park@sslab.ics.keio.ac.jp/, 2022.

[47] Keita Suzuki. scsi: qla2xxx: Fix double free in qla2x00_probe_one(). https://lore.kernel.org/lkml/20220426094031.750135-1-keitasuzuki.park@sslab.ics.keio.ac.jp/, 2022.

[48] Keita Suzuki. tracing: Fix potential double free in create_var_ref(). https://lore.kernel.org/lkml/20220426052921.2088416-1-keitasuzuki.park@sslab.ics.keio.ac.jp/, 2022.

[49] S M S Talebi, Z Yao, A A Sani, Z Qian, and others. Undo workarounds for kernel bugs. *30th USENIX Security*, 2021.

[50] BB Meshram V Shakti D Shekar and MP Varshapriya. Device driver fault simulation using KEDR. *International Journal of Advanced Research in Computer Engineering and Technology*, page 580–584, 2012.

[51] Vadim Fedorenko. net: decnet: fix netdev refcount leaking on error path. https://github.com/torvalds/linux/commit/3f96d644976825986a93b7b9fe6a9900a80f2e11, 2021.

[52] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 999–1010, New York, NY, USA, October 2020. Association for Computing Machinery.

[53] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, Hollywood, CA, October 2012. USENIX Association.

[54] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen Mc Camant, and Kangjie Lu. Understanding and detecting disordered error handling with precise function pairing. In *30th USENIX Security Symposium*, 2021.

[55] Wang X. io_uring: always let io_iopoll_complete() complete polled io. https://github.com/torvalds/linux/commit/dad1b1242fd5717af18ae4ac9d12b9f65849e13a5, 2022.

[56] Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. ARC++: effective typestate and lifetime dependency analysis. pages 116–126, July 2014.

[57] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 115–125, New York, NY, USA, 2005. Association for Computing Machinery.

[58] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. UBITect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 221–232, New York, NY, USA, November 2020. Association for Computing Machinery.

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