Path-Sensitive and Alias-Aware Typestate Analysis for Detecting OS Bugs

Tuo Li
Tsinghua University
Beijing, China

Jia-Ju Bai
Tsinghua University
Beijing, China

Yulei Sui
University of Technology Sydney
Sydney, Australia

Shi-Min Hu
Tsinghua University
Beijing, China

ABSTRACT

Operating system (OS) is the cornerstone for modern computer systems. It manages devices and provides fundamental service for user-level applications. Thus, detecting bugs in OSes is important to improve reliability and security of computer systems. Static typestate analysis is a common technique for detecting different types of bugs, but it is often inaccurate or unscalable for large-size OS code, due to imprecision of identifying alias relationships as well as high costs of typestate tracking and path-feasibility validation.

In this paper, we present PATA, a novel path-sensitive and alias-aware typestate analysis framework to detect OS bugs. To improve the precision of identifying alias relationships in OS code, PATA performs a path-based alias analysis based on control-flow paths and access paths. With these alias relationships, PATA reduces the costs of typestate tracking and path-feasibility validation, to boost the efficiency of path-sensitive typestate analysis for bug detection. We have evaluated PATA on the Linux kernel and three popular IoT OSes (Zephyr, RIOT and TencentOS-tiny) to detect three common types of bugs (null-pointer dereferences, uninitialized-variable accesses and memory leaks). PATA finds 574 real bugs with a false positive rate of 28%. 206 of these bugs have been confirmed by the developers of the four OSes. We also compare PATA to seven state-of-the-art static approaches (Cppcheck, Coccinelle, Smatch, CSA, Infer, Saber and SVF). PATA finds many real bugs missed by them, with a lower false positive rate.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; • **Security and privacy → Operating systems security**.

KEYWORDS

static analysis, operation system, bug detection

ACM Reference Format:
Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. 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 '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507770

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507770

1 INTRODUCTION

Operating system (OS) is the fundamental software of modern computer systems. Apart from classical general-purpose OSes (such as the Linux kernel), many new OSes have been developed for specific purposes. For example, due to the rise of IoT techniques, many IoT OSes (such as Zephyr) have been developed to manage IoT devices and support IoT applications. However, each OS inevitably has bugs, as it is quite large and complex. Even a simple OS bug (such as null-pointer dereference) can cause system crash, malicious attack and other runtime problems [72]. Thus, it is important to detect OS bugs to secure the foundation of computer systems.

Static typestate analysis [66] is a common technique to detect different types of bugs. Typestates associate state information with each program variable. This state information is used to determine which operations can be validly invoked upon a given variable. A *typestate property* is a finite state machine (FSM) to determine whether a sequence of observable operations are valid, and an invalid operation sequence can potentially cause a bug. Typestate analysis typically performs on top of the control-flow graph (CFG) of a program. To improve accuracy, some approaches [27, 29] perform path-sensitive analysis but focus on analyzing scalars not pointers. To solve this problem, some typestate approaches [32, 77] consider pointer alias relationships using imprecise flow-insensitive points-to analysis. Unfortunately, flow-insensitive alias results used in path-sensitive analysis can potentially introduce many false positives in bug detection, especially for large-size programs (like OSes) containing complex alias relationships.

Similar to typestate analysis, some generic static tools [8, 24, 25, 30, 55, 65] can detect different types of OS bugs based on predefined rules or variable states. Most of these approaches are path-insensitive (except CSA [25]) and use imprecise alias analysis (e.g., flow-insensitive analysis) or even ignore aliases, so they often report false positives and miss many real bugs.

To improve the accuracy of path-sensitive typestate analysis, it is important to capture precise alias relationships. However, there are two difficulties for analyzing OS code: *(D1)* Points-to analysis is insufficient to identify precise alias relationships in OSes. Generally, points-to analysis needs to model heap objects per memory allocation. However, due to the multi-module and application-driven

```
FILE: linux-5.6/drivers/media/platform/s5p-mfc/s5p_mfc.c
1266. static int s5p_mfc_probe(struct platform_device *pdev) {
        ......
1280.   dev->plat_dev = pdev;   // create alias relationship
1281.   if (!dev->plat_dev) {   // pdev can be NULL
1282.       dev_err(&pdev->dev, ...);  // Null-pointer dereference!
1283.       return -ENODEV;
1284.   }
        ......
1415. }
------------------------------------------------------------
// These interface functions have no explicit caller functions
// in the OS code
1664. static struct platform_driver s5p_mfc_driver = {
1665.   .probe = s5p_mfc_probe,     } Module interface functions
1666.   .remove = s5p_mfc_remove,   }
        ......
1672. }
```

Figure 1: A real null-pointer dereference in Linux 5.6.

nature of OSes, many functions do not have explicit caller functions. Thus, their pointer parameters can have incomplete points-to information, causing points-to analysis to miss many alias relationships. For example, `dev->plat_dev` and `pdev` in Figure 1 should be aliases and a null-pointer dereference at Line 1282 is triggered if the argument `pdev` is NULL. However, the function `s5p_mfc_probe` is implicitly called via a function-pointer field `.probe` of struct `s5p_mfs_driver` in another OS module. Thus, `pdev` has an empty points-to set, causing that `pdev` and `dev->plat_dev` are not treated as aliases since their points-to sets have no intersection. Therefore, the bug at Line 1282 cannot be found by points-to analysis based approaches. To handle such alias relationships, points-to analysis should record all the alias pairs generated by assignment statements, causing high memory overhead and unscalability to large codebases like OSes. *(D2)* An OS codebase is very large, containing an excessive number of variables and code paths. Thus, tracking variable typestates and checking path feasibility in OS code are expensive, especially when detecting multiple types of bugs.

Recently, some path-sensitive approaches [31, 63, 64, 69] conduct reachability analysis based on pre-computed value-flow graphs to detect specific types of bugs (e.g., memory leaks). But their value-flow graphs are built with points-to analysis, which can miss many alias relationships when analyzing OS code (*D1*). In addition, they perform only source-sink-based reachability analysis but not maintaining typestates, so they are not generic to multiple bug types.

Basic idea and novel techniques. Path-sensitive typestate analysis is effective in detecting bugs in applications, but applying this technique to OS code is challenging, because an OS typically has a large codebase and complex alias relationships. To solve this problem, our basic idea is: *(i1) identifying alias relationships based on control-flow paths and access paths without using points-to information, and (i2) using these alias relationships to reduce the costs of typestate tracking and code-path validation.* Based on this idea, we propose three novel techniques:

For *i1*, we propose a *path-based alias analysis* to compute alias relationships based on control-flow paths and access paths, without using points-to information. This analysis is inter-procedural, flow-sensitive and field-sensitive. For a control-flow path, this analysis maintains an alias graph at each program point to represent alias relationships in the path. Each alias graph is updated according to the analyzed instructions and access paths of the involved variables.

For *i2*, we observe that merging aliased variables can significantly reduce the number of typestates for bug detection and SMT constraints for path-feasibility validation, to boost analysis efficiency.

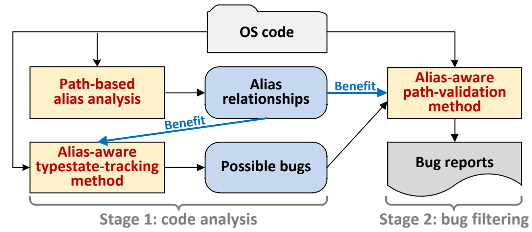


Figure 2: PATA workflow.

Based on this observation, we propose an *alias-aware typestate-tracking method* to efficiently detect multiple types of bugs, and an *alias-aware path-validation method* to efficiently check code-path feasibility of possible bugs. These two methods both benefit from the alias relationships identified by our path-based alias analysis.

Differences from existing approaches. First, unlike existing typestate-tracking methods [27, 29, 32, 77] that maintain one state for each variable, our alias-aware typestate-tracking method maintains one typestate for all variables in the same alias set, and updates this typestate when one of these aliased variables is handled by an instruction related to the target bug type. In this way, our method effectively reduces the amount of typestates that need to be tracked.

Second, unlike existing path-validation methods [31, 45, 64, 69] that build an SMT symbol for each variable to solve path constraints, our approach maps all variables in the same alias set to one SMT symbol to reduce the amount of SMT constraints to be solved. In addition, to accurately handle data structures, our typestate-tracking and path-validation methods are field-sensitive by distinguishing fields of a data structure.

Finally, unlike existing generic static tools [8, 24, 25, 30, 55, 65] for OS code, our alias-aware typestate-tracking method uses more alias relationships to improve accuracy, and our alias-aware path-validation method enables the path sensitivity of bug detection.

With the above three techniques, we develop PATA (**P**ath-sensitive and **A**lias-aware **T**ypestate **A**nalysis), a novel typestate analysis framework to detect OS bugs. PATA first identifies alias relationships without using points-to information and then uses these alias relationships to reduce the costs of typestate tracking and code-path validation. PATA has two stages shown in Figure 2. In Stage 1, PATA analyzes the OS code using our path-based alias analysis and alias-aware typestate-tracking method. For each code path, our alias analysis identifies alias sets as alias relationships; meanwhile, our alias-aware typestate-tracking method uses the identified alias relationships to analyze instructions in the code path to detect possible bugs, without validating path feasibility. In Stage 2, our path-validation method uses an SMT solver Z3 [85] to check the path feasibility of each possible bug to filter out false alarms, with the alias relationships identified in Stage 1. Finally, PATA produces readable reports of the found bugs. We have implemented PATA using LLVM [16] to automatically analyze OS code. Overall, we make four main contributions:

- We first analyze the challenges of path-sensitive typestate analysis for OS code, and then propose a new solution idea: (i1) identifying alias relationships based on control-flow paths and access paths without using points-to information, and (i2) using these alias relationships to reduce the costs of typestate tracking and code-path validation.

- Based on this idea, we propose three novel techniques: (1) a *path-based alias analysis* to identify alias relationships based on control-flow paths and access paths; (2) an *alias-aware typestate-tracking method* to effectively detect different types of bugs according to alias relationships; (3) an *alias-aware path-validation method* to efficiently filter out false bugs with an SMT solver and alias relationships. Note that typestate-tracking and path-validation methods both benefit from the alias relationships identified by path-based alias analysis.
- With the three techniques, we develop a novel path-sensitive and alias-aware typestate analysis framework named PATA, to effectively detect multiple types of OS bugs.
- We evaluate PATA on the Linux kernel and three popular IoT OSes (Zephyr, RIOT and TencentOS-tiny) to detect three common types of bugs (null-pointer dereferences, uninitialized-variable accesses and memory leaks). PATA finds 574 real bugs (including 463 null-pointer dereferences, 90 uninitialized-variable accesses and 21 memory leaks) with a false positive rate of 28%. 206 of these bugs have been confirmed by OS developers. We compare PATA to seven existing static approaches, and PATA finds many real bugs missed by them with a lower false positive rate.

2 MOTIVATION

2.1 A Motivating Example

Figure 3 shows a real null-pointer dereference in the Zephyr Bluetooth subsystem. In the function `friend_set`, the pointer `cfg` is first assigned with a data structure field `model->user_data` at Line 2709, and then it is compared to `NULL` in an *if* check at Line 2720, indicating that `cfg` and `model->user_data` can be `NULL`. If so, the function `send_friend_status` is called with `model` at Line 2748 in error handling code. In this function, the pointer `cfg` is assigned with the variable `model->user_data` at Line 2684. As `model->user_data` is `NULL` in this case, indicating that `cfg` is `NULL`, a null-pointer dereference can occur when `cfg->frnd` is accessed at Line 2687.

```
FILE: zephyr-2.1.0/subsys/bluetooth/cfg_srv.c
2680. static void send_friend_status(type *model, ...) {
      ......
2684.     struct bt_mesh_cfg_srv *cfg = model->user_data;  // Alias
      ......
2687.     net_buf_simple_add_u8(&msg, cfg->frnd);  // Unsafe dereference!
      ......
2692. }
------------------------------------------------------------
2705. static void friend_set(...) {
      ......
2709.     struct bt_mesh_cfg_srv *cfg = model->user_data;  // Alias
      ......
2720.     if (!cfg) {     // Pointer cfg can be NULL
2721.         BT_WARN(...);
2722.         goto send_status;
2723.     }
      ......
2747. send_status:
2748.     send_friend_status(model, ctx);
2749. }
```

Figure 3: A real null-pointer dereference in Zephyr.

This bug involves multiple alias relationships of data structure fields across multiple functions, and it is triggered only when `model->user_data` in the function `friend_set` is actually `NULL`. Such requirement is difficult to satisfy by executing existing test suites. In fact, this bug had existed for nearly 3 years since Zephyr 1.8.0 (released in Jun. 2017), and it was fixed by Zephyr developers based on a report generated by our PATA framework.

2.2 Challenges

Static typestate analysis has three important challenges when detecting bugs in OS code:

C1: Performing precise alias analysis. In OS code, due to the heavy use of pointers and data structure fields (like Figure 3), the alias relationships between variables can be very complex, especially when involving multiple code paths and function calls. Moreover, many OS functions do not have explicit caller functions in the OS code. Thus, their pointer parameters can have incomplete points-to information, making points-to analysis [1, 26, 35–37, 48, 49, 69, 82, 83] generally miss many alias relationships. Moreover, existing flow-sensitive must-alias or may-alias analyses [7, 40, 42, 43, 79, 88, 89] compute the intersection or union of alias sets at each joint points of different control-flow paths, which can miss many real alias relationships or introduce many false alias relationships for each control-flow path. Therefore, it is important to improve the precision of identifying alias relationships in OS code.

C2: Detecting multiple types of bugs. An effective typestate analysis framework should be applicable to multiple bug types by tracking the typestates of each variable. But there are lots of variables in the OS, and thus tracking the typestates of each variable can be quite expensive. Therefore, it is important to efficiently track typestates for multiple types of bugs.

C3: Dropping false bugs. On the one hand, without validating path feasibility, static typestate analysis often reports many false bugs. On the other hand, there are lots of code paths in the OS, and thus using an SMT solver to validate all possible code paths can be very costly. Therefore, it is important to check the feasibility of code paths with low costs.

3 KEY TECHNIQUES

To address the above challenges, we propose three key techniques. For *C1*, we propose a *path-based alias analysis* to identify alias relationships based on control-flow paths and access paths, without using points-to information. For *C2*, we propose an *alias-aware typestate-tracking method* to effectively detect different types of bugs according to alias relationships. For *C3*, we propose an *alias-aware path-validation method* to efficiently filter out false bugs with an SMT solver and alias relationships. We introduce them as follows.

3.1 Path-Based Alias Analysis

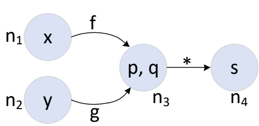
In OS code, a variable can be aliased with different variables in different control-flow paths. Thus, computing alias relationships for each control-flow path can produce precise alias results, which can effectively reduce false positives and negatives in bug detection. Moreover, each OS is modularly-designed and application-driven, causing that many functions do not have explicit caller functions in the OS code, and thus points-to sets of their pointer-type parameters can be incomplete. Based on these insights, we propose a *path-based alias analysis* by extending alias graph [43], and identify alias relationships according to control-flow paths and access paths, without using points-to information.

Alias graph. It is an important data structure to represent alias relationships in our alias analysis, so we introduce it first.

Definition 1. *An alias graph is a 2-tuple $G = \langle N, E \rangle$, where N is a set of nodes, and each node n represents an alias class (i.e., a set of variables Vars(n)) that points to one abstract object. E is a set of labeled edges. Each edge is labeled with a data structure field or a dereference operator "$*$", which represents how an abstract object is accessed.*

A variable residing in a node followed by a sequence of edge labels form an *access path* [13, 43]. Access paths ending with the same node on an alias graph form an *alias set*. Variables in the same alias set are aliases. Variables residing in a single node is considered as an access path with a length of 0.

Example 1. *Figure 4(a) shows an alias graph containing four nodes and three edges. Two edges are labeled with data-structure-field accesses (i.e.,*`f` *and* `g`*), and the other edge is labeled with a pointer dereference. Take node n_3 as an example, there are four access paths* `&x->f`, `&y->g`, `p` *and* `q` *to it, and the lengths of access paths* `p` *and* `q` *are both 0. The alias sets based on the access path results are shown in Figure 4(b).*



(a) Alias graph

AliasSet(n_1): {x}
AliasSet(n_2): {y}
AliasSet(n_3): {p, q, &x->f, &y->g}
AliasSet(n_4): {s, *p, *q, *(&x->f), *(&y->g)}

(b) Alias sets

Figure 4: Example of alias graph.

Given a node n and an edge label l, there is only one outgoing edge labeled with l from n. It indicates that a variable or an expression refers to only one abstract object per access path. Finally, every program point will maintain a separate alias graph based on a program path reaching this point. If Vars(n) of a node n changes during alias analysis, the alias graph is also considered as updated.

Building and updating alias graph. The alias graph is built from the entry of a function containing a set of isolated nodes, and each of them represents a single variable in the program. Then, our alias analysis updates the alias graph, according to the program instructions in form of the LLVM IR [50]. Our analysis focuses on four types of instructions that can handle alias relationships: MOVE($v_1 = v_2$), STORE($*v_2 = v_1$), LOAD($v_1 = *v_2$), and GEP($v_1 = \&v_2$->f). Note that our alias analysis is field-sensitive to handle data structures in OS code. Each access to a data structure field via LLVM's getelementptr instruction is handled through the GEP operation. The rules for each operation to update an alias graph are shown in Figure 5. The notations used in pseudocodes are described in Table 1. The four operations mentioned above are as follows:

HandleMOVE($v_1 = v_2$, G). After this operation, v_1 is represented by n_2 not n_1 (Lines 3-4), and thus v_1 and v_2 are represented by the same node, which indicates they become aliases. The change made on the variable sets of n_1 and n_2 indicates a changed alias graph.

HandleSTORE($*v_2 = v_1$, G). If n_2 has an outgoing edge labeled with $*$, it is dropped (Lines 8-9) and an edge labeled with $*$ from n_2 to n_1 is added (Line 11), so access paths $*v_2$ and v_1 reach the same node n_1. It indicates that after this operation, $*v_2$ and v_1 are aliases.

HandleLOAD($v_1 = *v_2$, G). If n_2 has an outgoing edge labeled with $*$, the target node of this edge represents v_1 after this operation (Lines 15-17). Otherwise, an edge labeled with $*$ from n_2 to n_1 (Line 19) is inserted. Thus, the access paths v_1 and $*v_2$ reach the same node, which indicates that v_1 and $*v_2$ are aliases.

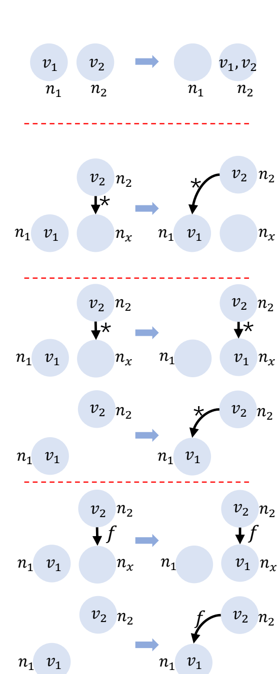
HandleGEP($v_1 = \&v_2$->f, G). This operation is similar to *HandleLOAD*, except that the edge is labeled with a data structure field f, instead of a dereference operator "$*$".

Table 1: Notation table of pseudocodes.

$n_i \in N$	A node in an alias graph $G = \langle N, E \rangle$.
$n_i \xrightarrow{l} n_j \in E$	A directed edge labeled in an alias graph. l is a field access or a pointer dereference, representing how an abstract object is accessed.
path	A stack of instructions (program statements) per control-flow path. It can also represent program point of the instruction on its top.
GetNode(v, N)	The node representing variable v.
Vars(n)	A set of variables that n represents.
GetArg(*func*, i)	The i_{th} formal parameter of *func*.
GetReturnValue(*func*)	The return value of *func*.
UpdateAliasGraph(*path*)	Alias-graph update under *path*.
TypestateTrack(*path*, G)	Bug detection given the current alias graph G and the code path *path* (This process will be introduced in Section 3.2).
Next(*inst*)	The successive instructions of *inst* on CFG.

```
define: HandleMOVE(v1 = v2, <N,E>)
 1: n1 := GetNode(v1, N);
 2: n2 := GetNode(v2, N);
 3: Vars(n1) := Vars(n1) − {v1};
 4: Vars(n2) := Vars(n2) ∪ {v1};
 5: return <N,E>;
define: HandleSTORE(*v2 = v1, <N,E>)
 6: n1 := GetNode(v1, N);
 7: n2 := GetNode(v2, N);
 8: if n2 -*-> nx ∈ E then
 9:    E := E − {n2 -*-> nx};
10: end if
11: E := E ∪ {n2 -*-> n1};
12: return <N,E>;
define: HandleLOAD(v1 = *v2, <N,E>)
13: n1 := GetNode(v1, N);
14: n2 := GetNode(v2, N);
15: if n2 -*-> nx ∈ E then
16:    Vars(n1) := Vars(n1) - {v1};
17:    Vars(nx) := Vars(nx) ∪ {v1};
18: else
19:    E := E ∪ {n2 -*-> n1};
20: end if
21: return <N,E>;
define: HandleGEP(v1 = &v2->f, <N,E>)
22: n1 := GetNode(v1, N);
23: n2 := GetNode(v2, N);
24: if n2 -f-> nx ∈ E then
25:    Vars(n1) := Vars(n1) - {v1};
26:    Vars(nx) := Vars(nx) ∪ {v1};
27: else
28:    E := E ∪ {n2 -f-> n1};
29: end if
30: return <N,E>;
```

(a) Rules for updating alias graph



(b) Examples of updating

Figure 5: Rules for updating alias graph.

Path-based alias analysis. For each control-flow path, it builds and updates alias graphs by analyzing each instruction in this path. Figure 6 shows the pseudocodes of our alias analysis. For each function without a caller function, the analysis builds an alias graph with each node represents a single variable in the OS code (Lines 1-6). Then, the analysis starts from the first instruction of the analyzed function (Lines 9-11), and performs a depth-first traversal along the control flow. The alias graphs are updated for each instruction (Lines 23-29). Bug detection is performed by tracking typestates of related alias set (Line 31). This process serves as an interface named *TypestateTrack*, which will be introduced in Section 3.2. To

```
define: AnalyzeCode()
 1: foreach func in OS code without a caller function do
 2:    G := ∅;
 3:    path := ∅;
 4:    foreach variable in the OS code do
 5:       insert a new node representing it into G;
 6:    end foreach
 7:    HandleFUNC(func, path, G);
 8: end foreach
define: HandleFUNC(func, path, G)
 9: inst := GetEntryOfFunc(func);
10: path.push(inst);
11: HandleINST(path, G);
define: HandleCALL(v = func(v_1, ..., v_n), path, G)
12: foreach i in 1..n do
13:    arg_i := GetArg(func, i);
14:    G' := HandleMOVE(arg_i = v_i, G);
15:    UpdateAliasGraph(path) := G';
16:    G := G';
17: end foreach
18: HandleFUNC(func, path, G);
19: ret := GetReturnValue(func);
20: G' := HandleMOVE(v = ret, G);
21: return G';
define: HandleINST(path, G)
22: inst := path.top();
23: switch typeof(inst):
24:    case v_1 = v_2: G' := HandleMOVE(inst, G); break;
25:    case *v_2 = v_1: G' := HandleSTORE(inst, G); break;
26:    case v_1 =*v_2: G' := HandleLOAD(inst, G); break;
27:    case v_1 = &v_2->f: G' := HandleGEP(inst, G); break;
28:    case v = func(v_1, ..., v_n): G' := HandleCALL(inst, path, G); break;
29: end switch
30: UpdateAliasGraph(path) := G';
31: TypestateTrack (path, G');
32: foreach inst' in Next(inst) do
33:    if inst' not in path then
34:       path.push(inst');
35:       HandleINST(path, G');
36:       path.pop();
37:    end if
38: end foreach
```

Figure 6: Pseudocodes of our path-based alias analysis.

avoid repeatedly handling loops and recursive calls, if a successive instruction is already handled in the path (with each loop and recursion unrolled only once), the analysis does not handle it again (Lines 32-38).

A function call is regarded as several MOVE operations between formal parameters and corresponding actual parameters (Lines 12-17), because they are aliases after passing parameters. Similarly, the return instruction of a callee function is also regarded as a MOVE operation (Lines 19-20).

Example 2. *We illustrate our path-based alias analysis with the simplified code in Figure 3, and present its alias graph for some important program points in Figure 7 (isolated nodes are omitted). Each node represents a set of aliased variables, and the nodes with bold edge are related to the branch condition at Line 4. We exploit* `func:v` *to represent the variable* `v` *in the function* `func`*. Through a GEP(*`foo:r=&foo:p->s`*) and LOAD(*`foo:t=*foo:r`*) operations, our analysis gets alias graph at Line 3 and infers that* `foo:t` *and* `*(&foo:p->s)` *are aliases. For the branch statement at Line 4, our analysis copies the alias graph into two branches. For example in the path ($Line_2, Line_3, Line_4, Line_5, Line_{10}, Line_{11}, Line_{12}$), the function* `bar` *is called at Line 5, and our analysis uses a MOVE(*`bar:p=foo:p`*) operation to pass related parameters. With a GEP(*`bar:r=&bar:p->s`*) and a LOAD(*`bar:t=*bar:r`*) operations, our analysis gets the alias graph at Line 11; and through a LOAD(*`bar:a=*bar:t`*) operation, our analysis gets the alias graph at Line 12.*

Referring to existing static approaches [5, 52, 64], to avoid spending too much time on analyzing loops and recursive calls, our alias analysis unrolls each loop and recursive call just once (Lines 32-38), which can miss some alias relationships in the two cases, causing soundness loss of bug detection.

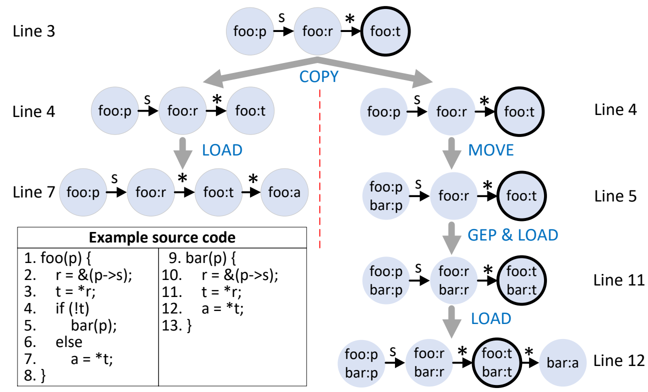


Figure 7: Example of illustrating path-based alias analysis.

3.2 Alias-Aware Typestate-Tracking Method

Static typestate analysis defines some "typestates" to describe possible states that each variable can reach, and then tracks typestate transitions according to related operations to detect bugs. But there are lots of variables and code paths in OS code, so tracking typestates for each variable and synchronizing typestates among aliased variables are quite expensive, especially when detecting multiple types of bugs. We consider merging aliased variables that may refer to the same memory location, so that their typestates can be merged to reduce analysis costs. Based on this consideration, we propose an *alias-aware typestate-tracking method* using the alias relationships produced by our path-based alias analysis, to detect multiple types of OS bugs. This method is represented as *TypestateTrack* in Figure 6.

Our method is field-sensitive, by regarding each field of a data structure as a separate variable in typestate tracking. It also considers alias relationships involving data structure fields, due to the field sensitivity of our alias analysis. Moreover, our method is interprocedural and flow-sensitive, but neglects the feasibility of code paths, and thus it can report some false bugs. To filter out these false bugs, we use a path-validation method, which will be introduced in Section 3.3.

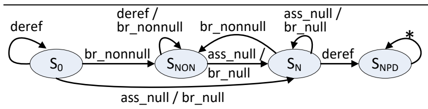
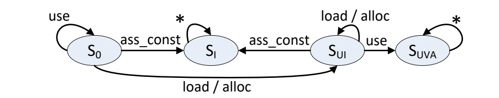
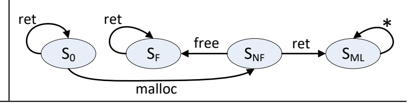
A typestate property for each variable can be specified as a finite state machine (FSM) [32].

Definition 2. *An FSM for detecting a specific type of bug is described as $FSM_{type} = \langle \sum, \mathbb{S}, S_0, \delta, S_{type} \rangle$, where:*

- *$\sum$ is the set of instructions that change the state.*
- *$\mathbb{S}$ is the set of all possible states.*
- *S_0 is the initial state.*
- *δ is a set of state-transition functions that map the present state and an instruction to a new state.*
- *S_{type} is the final state which means a possible bug is detected by TypestateTrack.*

For each code path, all aliased variables identified by our alias analysis share the same state in the FSM.

Table 2: FSMs of null-pointer dereferences (NPD), uninitialized-variable accesses (UVA) and memory leaks (ML).

$FSM_{NPD} = \langle \sum, \mathbb{S}, S_0, \delta, S_{NPD} \rangle$	$FSM_{UVA} = \langle \sum, \mathbb{S}, S_0, \delta, S_{UVA} \rangle$	$FSM_{ML} = \langle \sum, \mathbb{S}, S_0, \delta, S_{ML} \rangle$
$\mathbb{S} = \{S_0, S_{NON}, S_N, S_{NPD}\}$ S_{NON}. The alias set is non-NULL. S_N. The alias set is NULL.	$\mathbb{S} = \{S_0, S_{UI}, S_I, S_{UVA}\}$ S_{UI}. A local variable or a heap object is uninitialized. S_I. A local variable or a heap object is initialized.	$\mathbb{S} = \{S_0, S_{NF}, S_F, S_{ML}\}$ S_{NF}. A heap object is not freed. S_F. A heap object is freed.
$\sum = \{ass\_null, br\_null, br\_nonnull, deref\}$ ***ass_null***. Assign NULL to a pointer. ***br_null***. Execute a branch where the pointer is NULL. ***br_nonnull***. Execute a branch where the pointer is non-NULL. ***deref***. Dereference a pointer.	$\sum = \{ass\_const, load, alloc, use\}$ ***ass_const***. Assign a constant to a local variable or a heap object. ***load***. Load a value from an uninitialized heap object or an uninitialized data structure filed. ***alloc***. Load a local variable. ***use***. Access a variable or a heap object.	$\sum = \{malloc, free, ret\}$ ***malloc***. Allocate a heap object. ***free***. Free a heap object. ***ret***. Execute a return instruction.
		

Definition 3. *Function mapping an alias set AS to a corresponding state S in the FSM is defined as* $S^m : \mathbb{AS} \rightarrow \mathbb{S}$, *where* $\mathbb{AS}$ *represents the alias sets in the code path.*

Our typestate-tracking method and alias analysis are performed at the same time (Line 31 in Figure 6). For each instruction in the code path, after the alias graph G is updated, our method first finds the alias set AS of the variable handled by the analyzed instruction, with G, and gets the current state $S_{curr} = S^m(AS)$. Then, our method changes the state of related alias set, according to S_{curr} and the analyzed instruction. For different types of bugs, their FSMs can be separately maintained at the same time during typestate tracking.

At present, we have implemented three FSMs to detect null-pointer dereferences (NPD), uninitialized-variable accesses (UVA) and memory leaks (ML), respectively, because these three types of bugs are common and dangerous in OSes. The definitions of these FSMs are shown in Table 2. We use state-transition diagram to illustrate each state-transition function δ and use "$*$" to represent any input to FSM.

Example 3. *We use an example in Figure 8 to illustrate how to simplify typestate tracking with alias relationships. Without alias relationships, to detect null-pointer dereference in Figure 7, typestate analysis maintains states for* `foo:t` *and* `bar:t` *separately, and transfers its state to* NULL *when analyzing the variable* `bar:t` *at Line 11, because the state of its aliased variable* `foo:t` *is* NULL. *The related state transitions are shown in Figure 8(a). Instead, with alias relationships, our method merges states of aliased variables to simplify typestate tracking. In Figure 8, our method maintains just one state for the alias set of* `foo:t` *and* `bar:t`, *because these variables become aliases and share the same state. The related alias-aware state transitions are shown in Figure 8(b). Comparing Figure 8(a) and Figure 8(b), we find that our method can effectively simplify state transitions and thus reduce the cost of typestate tracking, by using alias relationships.*

Due to unsoundness of our alias analysis when handling loops and recursive calls, our typestate-tracking method may miss the opportunity to merge the states of some aliased variables. Moreover, without validating code-path feasibility in alias analysis, our typestate-tracking method may mistakenly merge the states of two variables referring to different memory locations, which can introduce inaccuracy of bug detection.

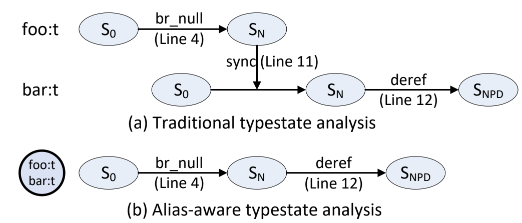


Figure 8: Bug-related state transitions in Figure 7.

3.3 Alias-Aware Path-Validation Method

On the one hand, we observe that the code paths of possible bugs often occupy a small proportion of all code paths in the whole OS code, and thus validating all code paths are redundant in bug detection. On the other hand, we observe that all aliased variables should satisfy the same constraints in a given code path, and thus these variables can be represented by the same symbol in the SMT solver, to reduce the cost of path constraint solving. Based on the two observations, we propose an *alias-aware path-validation method* using the alias relationships produced by our path-based alias analysis, to efficiently filter out false bugs reported by our typestate-tracking method. Besides, this method is field-sensitive, by regarding each field of a data structure as a separate variable in path validation. Due to the field sensitivity of our alias analysis, this method considers alias relationships involving data structure fields in path validation.

In our method, constraints in path validation are simplified by mapping an alias set not a variable to one symbol in an SMT solver. During path validation, if the symbol does not exist, our method creates a new symbol for the alias set.

Definition 4. *Function mapping an alias set AS to a symbol X in an SMT solver is defined as* $X^m : \mathbb{AS} \rightarrow \mathbb{X}$, *where* $\mathbb{AS}$ *are alias sets in the code path, and* $\mathbb{X}$ *are SMT symbols.*

To validate the code-path feasibility of each possible bug, our method translates the instructions in its code path to SMT constraints, and then uses the SMT solver Z3 to compute whether these constraints can be satisfied. Specifically, for each instruction, our method first gets the alias set of the handled variable, then finds the symbol of this alias set, and finally builds constraints for this symbol with instruction information.

Definition 5. *Function to get the symbol X for the variable v is defined as* $R(v) = X^m(AS)$ *where v is in the alias set AS.*

Specifically, when building constraints, we formulate each instruction in the following tiny source language:

Table 3: Translation rules of expressions and instructions.

Source	SMT constraints
(a) Translation of L-values	
$T_{var}(v)$ where $v \in \langle var \rangle$	$R(v)$
(b) Translation of expressions	
$T_{exp}(c)$ where $c \in \langle const \rangle$	c
$T_{exp}(var)$ where $v \in \langle var \rangle$	$T_{var}(v)$
$T_{exp}(e_1 op_b e_2)$	$T_{exp}(e_1)\ op_b T_{exp}(e_2)$
$T_{exp}(op_u e_1)$	$op_u T_{exp}(e_1)$
(c) Translation of statements	
$T_{stm}(var := e)$	$T_{var}(var) == T_{exp}(e)$
$T_{stm}(brt(e))$	$T_{exp}(e) == 1$
$T_{stm}(brf(e))$	$T_{exp}(e) == 0$

- $\langle exp \rangle ::= \langle const \rangle \mid \langle var \rangle \mid \langle exp \rangle_1\ op_b\ \langle exp \rangle_2 \mid op_u\ \langle exp \rangle$
- $\langle stm \rangle ::= \langle var \rangle = \langle exp \rangle \mid brt(e) \mid brf(e)$

In the source language, *expr* represents an expression like `a+1`; *const* represents a constant value; *var* represents a variable; op_b represents a binary operator; op_u represents a unary operator; *stm* represents a statement such as `v=a+1`; $brt(e)$ represents a condition to execute a control-flow branch when e is evaluated to be true (e.g., `if(e)`); $brf(e)$ represents the condition when e is evaluated false. Translation rules from a source language to SMT constraints are shown in Table 3.

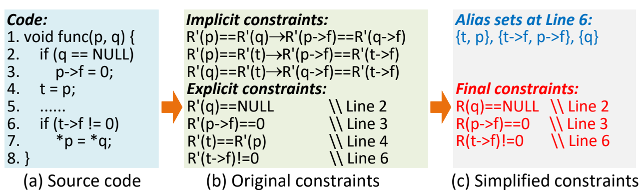


Figure 9: Example of simplifying SMT constraints.

If the conjunction of these SMT constraints is satisfiable, the validated code path is considered to be feasible, and thus the corresponding possible bug is identified to be real.

Example 4. *We illustrate how to use alias relationships to simplify SMT constraints, using an example in Figure 9 (type information is omitted). To validate the code path of a possible null-pointer dereference ($Line_2, Line_3, Line_4, Line_6, Line_7$) in Figure 9(a), we need to translate the instructions in the code path to SMT constraints. Suppose the function* `R'()` *maps a variable to an SMT symbol without considering alias relationships, for each assignment like* `p1=p2`*, we need to add an explicit constraint* `R'(p1)==R'(p2)`*. If* `p1` *and* `p2` *are data structure pointers of the same type, each of their field* `f` *should be equal. Thus, we need to add an implicit constraint* `R'(p1)==R'(p2)→R'(p1->f)==R'(p2->f)`*, where → means implication. Figure 9(b) shows the constraints without considering alias relationships. Instead, by considering alias relationships, if two variables* `p1` *and* `p2` *becomes aliases, our method maps them to the same SMT symbol (*Definition 5*), causing that* `R(p1)==R(p2)` *is naturally satisfied, and thus this explicit constraint can be dropped. If two variables* `p1` *and* `p2` *become aliases, their fields like* `p1->f` *and* `p2->f` *can be also inferred to be aliases, causing that these fields are mapped to the same SMT symbol and implicit constraints like* `R(p1)==R(p2)→R(p1->f)==R(p2->f)` *are naturally satisfied, and thus these implicit constraints can be also dropped. Figure 9(c) shows the alias sets used for constraint simplification and the simplified constraints. In this example,* `R(p->f)==0` *and* `R(t->f)!=0` *cannot be satisfied at the same time, so this possible bug is identified to be false.*

Due to unsoundness of our alias analysis when handling loops and recursive calls, our method may lose some constraints about multiple executions of loop body and recursive function, and thus can cause false positives in bug detection.

4 FRAMEWORK

Based on the three key techniques in Section 3, we develop a novel path-sensitive and alias-aware typestate analysis framework named PATA, to effectively detect multiple types of OS bugs. We implement PATA using Clang 9.0 [16]. Figure 10 shows the architecture of PATA, which has three phases:

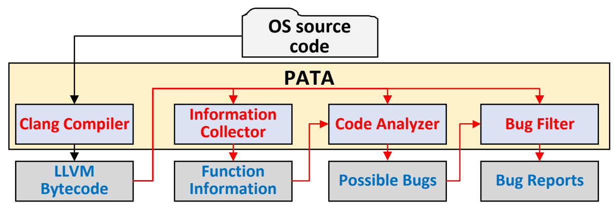


Figure 10: PATA architecture.

P1: Code compilation and code-information collection. The Clang compiler compiles the OS source code into LLVM bytecode, and then the information collector scans each LLVM bytecode file to record function information (including the position of each function definition and function name, etc.) in a database. Such information is used in subsequent code analysis for inter-procedural analysis across source files.

P2: Code analysis. The code analyzer uses our path-based alias analysis and alias-aware typestate-tracking method to analyze LLVM bytecode files, without validating path feasibility. The analysis starts at the entry of each function without explicit callers, and handles each code path in top-down analysis. When a function returns, the analysis combines the information of its code paths to mitigate path explosion. Finally, the analysis produces possible bugs with their code paths.

P3: Bug filtering. For a given real bug, there may be multiple code paths between its two problematic instructions, and thus many repeated bugs can be reported. To drop repeated bugs, for a new possible bug, the bug filter checks whether its problematic instructions are identical to those of any already detected bug. If so, this possible bug is considered to be repeated and thus dropped. Then, the bug filter uses our alias-aware path-validation method to drop false bugs.

False positives. PATA can still report false bugs due to the limitations of current implementation. For example, PATA unrolls each loop and recursive call just once, so it can report false bugs involving multiple executions of loop body and recursive function. Moreover, PATA does not handle non-constant array indexes, data dependence across functions with a variable number of parameters or concurrency of memory accesses, so it can report false bugs related to these aspects.

5 EVALUATION

We evaluate PATA on the Linux kernel and three open-source IoT OSes (Zephyr [86], RIOT [59] and TencentOS-tiny [73]). Table 4 shows their information, and source code lines are counted by CLOC [17]. For the Linux kernel, we use the kernel configuration *allyesconfig* to enable all kernel code for the x86-64 architecture. For each IoT OS, many source files are architecture-specific, so we have tried our best to compile as many source files as possible, by tuning available compilation configurations. We run the evaluation on a regular x86-64 desktop with eight processors and 16 GB memory.

Table 4: Information about the four checked OSes.

OS	Version	Source files (*.c)	LOC
Linux kernel	5.6	28,260	14.2M
Zephyr	2.1.0	1,669	383K
RIOT	2020.04	4,402	1,575K
TencentOS-tiny	Commit 23313e	1,497	572K

5.1 Bug Detection

We run the three checkers implemented in Section 3.2 to detect null-pointer dereferences (NPD), uninitialized-variable accesses (UVA) and memory leaks (ML). Each checker is implemented with just 100-200 lines of code. We manually check all the bugs found by PATA. Table 5 shows the results.

Code analysis. PATA in total analyzes 10.3M lines of code in 18.4 K source files. The remaining 6.5M lines of code in 17.4K source files are not analyzed, as they are not enabled by the compilation configurations used by us. We believe that PATA can find more bugs, if these source files can be compiled with proper configurations. Moreover, compared to alias-unaware typestate tracking and path validation, PATA drops 49.8% typestates and 87.3% SMT constraints, which effectively reduces the complexity and costs of static analysis. Finally, PATA drops 54.7 K false bugs using our path-validation method, which effectively improves bug-detection accuracy.

Found bugs. PATA reports 797 bugs, and a PhD student spent 12 hours on checking the bug reports. This time usage is smaller than what we expected, as some reported bugs have similar root causes or patterns and they can be checked together. Finally, we identify that 574 of them are real bugs, including 463 null-pointer dereferences, 90 uninitialized-variable accesses and 21 memory leaks. Thus, the overall false positive rate of bug detection is 28%. In our experience, reporting too many bugs within a short time is not recommended by the Linux community. Thus, similar to existing works [4, 5], we randomly selected 200 real bugs in Linux kernel and all the 120 real bugs in IoT OSes, and sent them to OS developers. 206 of them (138 in Linux, 23 in Zephyr, 23 in RIOT and 22 in TencentOS-tiny) have been confirmed. We are still waiting for the response of the remaining bugs. Besides, 13 of our patches that fix 46 bugs have been applied in the OS code, and the 160 remaining confirmed bugs have been fixed by OS developers according to our bug reports.

Bug distribution. We classify the 574 real bugs found by PATA, by the category of the OS part containing the bug. Figure 11 shows the bug distribution. We find that drivers have 75% of the real bugs in the Linux kernel, and third-party modules have 68% of the real bugs in the three IoT OSes. Indeed, many Linux drivers and all third-party IoT OS modules are developed by third-party organizations not the OS community, and their code quality are generally worse than that of other OS parts [20]. In addition, we find that network modules and filesystems have 16% of the real bugs in the Linux kernel, and subsystem modules (including network stacks, bluetooth modules, etc) have 25% of the real bugs in the three IoT OSes. As these OS parts are commonly-used and security-critical, their bugs are often dangerous and received serious attention by OS developers after we reported them.

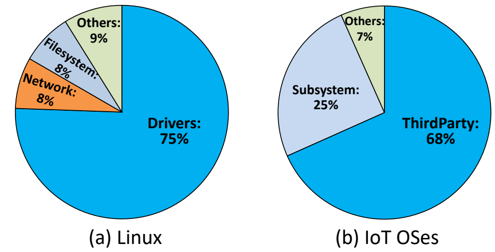


Figure 11: Distribution of the found bugs.

5.2 False Positives

PATA still reports 223 false bugs in the four OSes, and these false bugs are introduced for three main reasons:

First, PATA is array-insensitive and thus inaccurate in handling array elements with non-constant array indexes. For example, PATA identifies that two array elements `array[i+1]` and `array[j]` are different, even if the statement "`j=i+1`" is placed before the accesses to them, because their access paths in our alias analysis are different.

Second, although PATA uses Z3 to validate path feasibility, it still errs in handling some complex cases, such as complex arithmetic conditions and data dependence across multiple functions. PATA also fails to check loop conditions for multiple iterations and thus can report false bugs involving loops.

Finally, PATA neglects the concurrency of memory accesses. For example, the initialization and access to a variable can be respectively performed in two concurrently-executed functions with synchronization, which guarantees that the initialization is always performed before the access. But when analyzing the access, PATA may fail to find any initialization to this variable before the access due to thread unawareness, and thus it can report a false uninitialized-variable access.

5.3 Case Studies of Bugs Found by PATA

Figure 12 shows several real bugs found by PATA, and these bugs have been confirmed and fixed by OS developers.

Null-pointer dereferences in Linux MCDE driver. In Figure 12(a), the variable `d->mdsi` is compared with `NULL` at Line 1035 in the function `mcde_dsi_bind`, namely this variable can be `NULL`. Then, the function `mcde_dsi_start` is called at Line 1064. In this function, `d->mdsi` is dereferenced at Lines 724, 752, 778 and 787, which can cause null-pointer dereferences. To fix these bugs, the developer drops the call to `mcde_dsi_start` when `d->mdsi` is NULL.

Null-pointer dereference in Zephyr IP network stack. In Figure 12(b), the variable `dst_addr` is compared with `NULL` at Line 1361 in the function `context_sendto`, namely this variable can be `NULL`. At Line 1361, when `dst_addr` is `NULL` and `msghdr` is not

Table 5: Analysis results of the four OSes.

Description		Linux	Zephyr	RIOT	TencentOS-tiny	*Total*
Code analysis	Source files (analyzed/all)	16,237/28,260	634/1,669	1,134/4,402	398/1,497	18.4K/35.8K
	Source code lines (analyzed/all)	9,539K/14,223K	254K/383K	374K/1,575K	180K/572K	10.3M/16.8M
	Typestates (alias-aware/unaware)	22,016M/43,981M	249M/437M	699M/1,261M	51M/81M	23.0G/45.8G
	SMT constraints (alias-aware/unaware)	238M/1,903M	1,302K/3,926K	3,685K/11,014K	1,050K/2,071K	244M/1,920M
Bug detection	Dropped repeated bugs	18,354K	220K	143K	111K	18.8M
	Dropped false bugs	48,472	3,884	1,514	873	54.7K
	Found bugs (NPD/UVA/ML)	627 (508/102/17)	30 (27/2/1)	106 (98/5/3)	34 (14/13/7)	797 (647/122/28)
	Real bugs (NPD/UVA/ML)	454 (365/76/13)	24 (24/0/0)	67 (62/2/3)	29 (12/12/5)	574 (463/90/21)
	Confirmed bugs (NPD/UVA/ML)	138 (94/31/13)	23 (23/0/0)	23 (20/0/3)	22 (5/12/5)	206 (142/43/21)
Time usage		33h01m	44m	82m	22m	35h29m

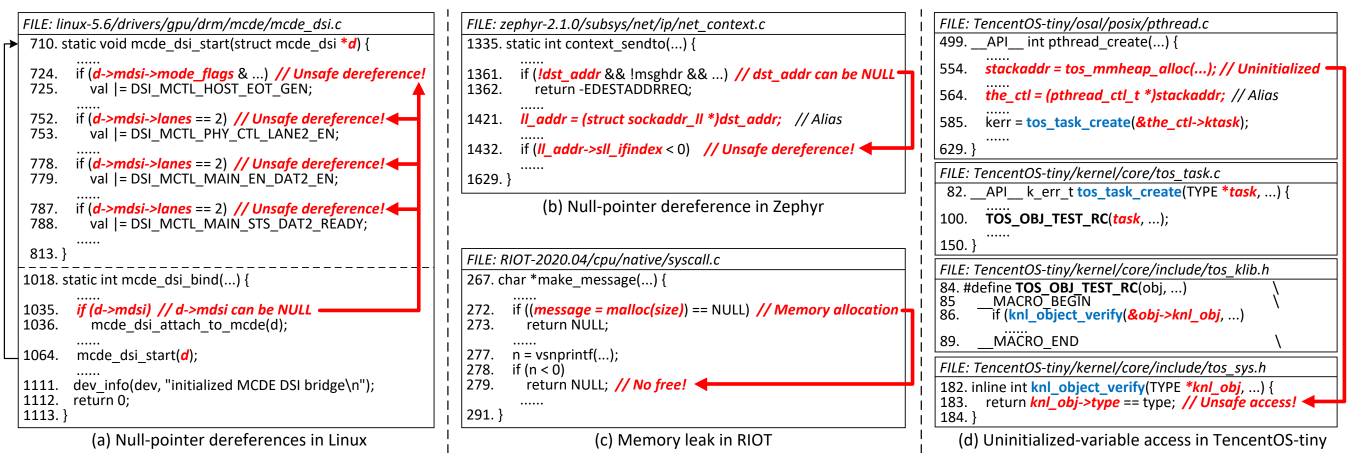


Figure 12: Example bugs found by PATA.

NULL, the function does not return at Line 1362 and continues execution. Then, dst_addr is assigned to ll_addr at Line 1421, and thus ll_addr can be NULL. After that, ll_addr is dereferenced at Line 1432, causing a null-pointer dereference. To fix this bug, the developer refactored the source code in the function context_sendto to handle the case that dst_addr is NULL.

Memory leak in RIOT syscall-handling component. In Figure 12(c), the variable message points to a memory area allocated by calling malloc at Line 272 in the function make_message. Then, it returns due to an exception at Line 279, without releasing the memory area pointed by message, causing a memory leak. To fix this bug, the developer calls free(message) before the return statement at Line 279, to free the allocated memory in error handling.

Uninitialized-variable access in TencentOS-tiny thread library. In Figure 12(d), the variable stackaddr points to an uninitialized memory area allocated by tos_mmheap_alloc at Line 554 in the function pthread_create. After that, stackaddr is assigned to the_ctl at Line 564, and the function tos_task_create is called with &the_ctl->ktask at Line 585. Finally, via two function calls and a macro, the variable (the_ctl->ktask).knl_obj.type is accessed at Line 183 in the function knl_object_verify. But the memory area pointed by the_ctl is uninitialized, causing an uninitialized-variable access here. To fix this bug, the developer calls memset to initialize the memory area pointed by stackaddr after calling tos_mmheap_alloc.

5.4 Sensitivity Analysis

The core idea of PATA is to exploit alias relationships to enhance typestate analysis for OS code. To validate the value of this idea, we implement a non-alias version of PATA, named *PATA-NA*, which does not compute alias relationships in typestate analysis. Table 6 shows the results in Linux.

Table 6: Sensitivity analysis results in Linux.

Description	PATA-NA	PATA
Found Bugs (NPD/UVA/ML)	620 (424/108/88)	627 (508/102/17)
Real Bugs (NPD/UVA/ML)	194 (168/15/11)	454 (365/76/13)
Time usage	8h19m	33h01m

PATA-NA finds 620 bugs in Linux and 194 of them are real, achieving a false positive rate of 69% that is higher than PATA. These 194 real bugs are all found by PATA, and PATA additionally finds 260 bugs missed by PATA-NA. Moreover, PATA spends less time than PATA-NA, by merging typestates and SMT constraints according to alias relationships. The results indicate that using alias relationships in typestate analysis indeed improves the accuracy and efficiency of bug detection.

Table 7: Bugs found by three additional checkers in Linux.

Bug type	*Double lock/unlock*	*Array index underflow*	*Division by zero*	*Total*
Found bugs	22	23	7	52
Real bugs	18	20	5	43

5.5 Generality to Other Bug Types

Benefiting from typestate analysis, PATA can conveniently detect different types of OS bugs with different checkers. To validate such generality, we also implement three additional checkers to detect other three common types of OS bugs, including double-lock/unlock, array-index-underflow and division-by-zero bugs. Each of these checkers is implemented according to its bug-related FSM and using just 100-200 lines of code, like the three checkers used in Section 5.1. Table 7 shows the results of these additional checkers in Linux.

With these additional checkers, PATA additionally finds 52 bugs, and we identify that 43 of them are real bugs, including 18 double-lock/unlock, 20 underflow and 5 division-by-zero bugs. The results indicate the generality of PATA to different types of OS bugs.

6 COMPARISON TO EXISTING APPROACHES

We experimentally compare PATA to seven state-of-the-art static analysis approaches, including Cppcheck [24] (v2.3), Coccinelle [55] (v1.0.8), Smatch (v0.5.0) [65], CSA (checker-279) [25], Facebook Infer [39] (v1.1.0), Saber [69] (v2.1) and SVF [67] (v2.1). Cppcheck, Coccinelle, Smatch, CSA and Infer are open-source static analysis tools that can detect multiple types of bugs; Saber is a path-sensitive static analysis tool to detect memory leaks; SVF is a static value-flow analysis framework that contains a flow-sensitive and inter-procedural points-to analysis, which can be used to detect bugs.

For Cppcheck, Smatch, CSA and Infer, we use them to detect the three types of bugs detected by PATA in Section 5.1; For Coccinelle, we just use its existing semantic patches [61] to detect null-pointer dereferences, as we do not find any existing semantic patch to detect uninitialized-variable accesses or memory leaks; For Saber, we use it to detect memory leaks. For SVF, we replace the path-based alias analysis with the SVF's flow-sensitive points-to analysis in PATA, to implement a new tool named SVF-Null to detect null-pointer dereferences. To evaluate Saber and SVF-Null, we use WLLVM [76] to build the whole Linux kernel into a single LLVM bytecode file as SVF wiki [70] suggests, and use SVF-Null to perform analysis on the bytecode file. But we fail to build the three IoT OSes using WLLVM due to many compilation errors, and thus we use Saber and SVF-Null to analyze bytecode files generated by Clang for each single source file. Note that Smatch and CSA report many compilation errors when checking IoT OSes, as their compilation scripts are unsuitable to the Makefiles of IoT OSes. Similarly, Infer reports many compilation errors when checking the Linux kernel. Besides, because the whole Linux kernel has lots of pointers, Saber and SVF consume too much memory when checking its code, and finally abort due to insufficient memory. Similarly, several recent works [31, 64] also find that Saber and SVF can consume too much memory or time when checking large-scale programs. For the above reasons, we use Smatch and CSA to just check the Linux kernel, and use Infer, Saber and SVF to just check the three IoT OSes. Table 8 shows the detailed comparison results of these approaches:

(1) 27 real bugs found by Cppcheck, 6 real bugs found by Coccinelle, 110 real bugs found by Smatch, 196 real bugs found by CSA, 15 real bugs found by Infer, 2 bugs found by Saber and 4 bugs found by SVF-Null are also found by PATA. But 25 real bugs found by Cppcheck and 2 real bugs found by Coccinelle are missed by PATA. Indeed, the source files containing the 27 missed bugs are not compiled with the compilation configurations used in our evaluation, so these source files are not checked by PATA; while Cppcheck and Coccinelle check source files without code compilation. We believe if these source files can be compiled with proper configurations, the 27 missed bugs can be also found by PATA.

(2) PATA finds 328 real bugs missed by the seven tools (note that some bugs found by these tools are identical) with a lower false positive rate. Due to lacking inter-procedural analysis or alias analysis, Cppcheck, Coccinelle and Smatch miss complex bugs involving multiple functions or alias relationships. Moreover, the three tools do not validate code path feasibility, and thus they report many false bugs caused by infeasible code paths. Though CSA, Infer, Saber and SVF-Null compute points-to information to handle alias relationships, their points-to analyses fail to model heap objects for pointer parameters of module interface functions and miss complex alias relationships in specific code paths, and thus these tools miss many real bugs related to pointer parameters and report many false bugs involving complex alias relationships. In addition, Infer and Saber fail to handle some complex path conditions especially those related to return values of callee functions, and thus they also report some false bugs.

(3) PATA spends more time than Cppcheck, Coccinelle, Smatch, CSA, Saber and SVF-Null in code analysis, as it computes alias relationships more precisely and performs path-sensitive analysis. Even so, PATA finds many more real bugs, so we believe that the effectiveness of its bug detection outweighs in its time overhead. PATA spends less time than Infer, due to its efficient analysis techniques, such as alias-aware typestate tracking and path validation.

Other approaches. Besides the above seven open-source approaches, there are some other OS-bug detection approaches that detect specific bug types or are closed-source. For example, UBITect [87] targets use-before-initialization bugs in OS code, and it performs source-sink analysis and searches for a feasible path between the source (allocation site) and the sink (use site) using symbolic execution; while PATA first performs alias-aware typestate analysis without checking code-path feasibility, and then it uses alias relationships to efficiently check the code-path feasibility of each possible bug. MLEE [75] focuses on early-exit paths and detects memory leaks by comparing these paths to normal paths in OS code; while PATA considers more code paths and can detect memory leaks via typestate tracking. Moreover, when identifying alias relationships, both UBITect and MLEE use points-to analysis that can introduce some inaccuracy, while PATA performs path-based alias analysis that can be more accurate. Coverity [21] is a commercial static analysis tool that can detect different kinds of bugs. Linux and Zephyr developers use it to check their code before each OS version is released [22, 23]. Thus, we believe that the bugs found by PATA in these two OSes should be missed by Coverity.

7 DISCUSSION

Benefiting other analyses with alias analysis. We believe that the path-based alias analysis in PATA can be used to boost the performance of other types of analysis. For example, in symbolic execution [12, 56, 81], aliased variables can be mapped into a single symbol with this alias analysis, to merge many constraints among these variables, which can simplify constraint solving with no or small precision loss. In model checking [11, 53], aliased variables in a program can be mapped into a single variable in the model, to reduce the state space of the checked model, which can mitigate state explosion problem. In API-rule checking [84], alias information can

Table 8: Comparison results of the four OSes.

OS bug detection		Cppcheck (NPD/UVA/ML)	Coccinelle (NPD)	Smatch (NPD/UVA/ML)	CSA (NPD/UVA/ML)	Infer (NPD/UVA/ML)	Saber (ML)	SVF-Null (NPD)	PATA (NPD/UVA/ML)
Linux	Found bugs	324 (157/154/13)	35	423 (194/204/25)	1,151 (848/283/20)	-	*OOM*	*OOM*	627 (508/102/17)
	Real bugs	51 (44/6/1)	6	110 (87/19/4)	196 (156/40/0)	-	*OOM*	*OOM*	454 (365/76/13)
	Time usage	3h34m	13h40m	17h15m	19h32m	-	*OOM*	*OOM*	33h01m
Zephyr	Found bugs	8 (1/7/0)	0	-	-	44 (16/28/0)	4	14	30 (27/2/1)
	Real bugs	1 (1/0/0)	0	-	-	1 (1/0/0)	0	0	24 (24/0/0)
	Time usage	24s	69s	-	-	197m	16s	54s	44m
RIOT	Found bugs	49 (14/33/2)	2	-	-	54 (26/26/2)	9	11	106 (98/5/3)
	Real bugs	6 (6/0/0)	2	-	-	10 (8/1/1)	2	1	67 (62/2/3)
	Time usage	57s	201s	-	-	166m	5s	67s	82m
TencentOS-tiny	Found bugs	63 (2/36/25)	2	-	-	46 (24/22/0)	8	3	34 (14/13/7)
	Real bugs	3 (2/1/0)	0	-	-	4 (3/1/0)	0	3	29 (12/12/5)
	Time usage	14s	46s	-	-	32m	13s	23s	22m

help to detect hard-to-find API misuses (e.g., caused by improper or wrong uses of arguments) involving complex alias relationships.

Limitations of PATA. PATA still has several limitations in detecting OS bugs. For example, PATA does not handle function-pointer calls, and thus it cannot find bugs whose bug-trigger paths passing through indirect function calls. Thus, we plan to introduce existing function-pointer analysis [51, 54] in PATA. In addition, To reduce the complexity of analyzing loops and recursive calls in our static analysis, we unroll each loop and recursive call just once, which can also cause unsoundness with reduced the accuracy of our bug detection. Thus, we plan to adapt some loop-oriented approaches [33, 68] to handle complex cases involving loops and recursions.

8 RELATED WORK

8.1 Static Analysis

Alias analysis. Many existing approaches [1, 9, 10, 26, 35–37, 48, 49, 67, 82, 83] perform points-to analysis and identify two pointers to be aliases if their points-to sets have variables in common. These approaches require all pointers to be initialized, so the points-to sets of these pointers are not empty. To compute alias relationships without points-to information, some approaches [7, 28, 38] perform alias analysis based on access paths. Kastrinis et al. [43] design an efficient data structure named alias graph to represent access path, for flow-sensitive but path-insensitive must-alias analysis of Java.

Typestate analysis. Some approaches [2, 29, 34, 46, 74] use typestate analysis to detect various types of bugs in applications. Hallem et al. [34] design a typestate analysis framework named *xgcc* with a flexible language named *metal* to define typestate transitions for bug detection. However, *xgcc* neglects alias relationships, so it is limited in tracking typestates involving complex alias relationships. To solve this problem, some approaches [27, 32, 77, 80] identify alias relationships with flow-insensitive pointer analysis. However, they introduce many false positives due to identifying imprecise alias relationships. Several approaches [3, 78] use precise on-demand backward-alias analysis to improve the accuracy of typestate analysis, but they can only detect specific bugs about variable tainting.

Value-flow analysis. Some approaches [31, 63, 64, 69] use value-flow analysis to detect bugs in applications. They exploit def-use chains to build value-flow graphs (VFG) [14, 69], and detect bugs by solving source-sink problems on the graphs. To improve the accuracy of bug detection, these approaches compute points-to information to identify alias relationships. But many OS functions do not have explicit caller functions, so their pointer parameters have incomplete points-to information, causing that points-to analysis can miss many alias relationships. As a result, these approaches can have many false positives and negatives when checking OS code.

Generic bug detection in OS code. Several static tools [8, 24, 25, 30, 55, 65] can detect different types of bugs in OS code. But their alias analysis is imprecise (due to flow insensitivity) or even lacked, and most of them (except CSA [25]) are path-insensitive in code analysis. Thus, these tools often report false positives and miss many real bugs.

Advantages of PATA. First, different from existing alias analysis, PATA identifies alias relationships in the OS code according to control-flow paths and access paths, without points-to information. Second, PATA is path-sensitive to effectively reduce false positives. Finally, PATA strategically uses alias relationships to reduce the complexity and costs of typestate tracking and code-path validation.

8.2 Symbolic Execution

Some approaches [12, 15, 19, 47, 57, 58] use symbolic execution to check the OS code. KLEE [12] is a well-known symbolic execution engine implemented with LLVM. It explores possible execution paths with constraint solving and generates concrete test cases for each path. But symbolic execution is often time consuming in analyzing large programs, because it needs to explore numerous code paths and solve their path constraints with an expensive SMT solver. To reduce time cost of solving path constraints, PATA merges SMT constraints involving aliased variables. Moreover, PATA only validates the feasibility of the code paths for possible bugs, instead of all possible code paths during static analysis.

8.3 Dynamic Analysis

Dynamic analysis has been widely used to detect OS bugs at runtime. Some approaches [20, 44, 60, 71] use coverage-guided fuzzing to test infrequently-executed code, by automatically mutating and generating system calls according to code coverage. Some approaches [6, 18, 41, 62] perform software fault injection to test error handling code, by deliberately corrupting the return values of kernel-interface calls. By using exact runtime information about OS execution, dynamic analysis can effectively reduce false positives in bug detection. However, dynamic analysis requires substantial test cases to achieve high code coverage and reduce false negatives, and it also degrades OS performance caused by runtime monitoring.

9 CONCLUSION

In this paper, we develop a novel path-sensitive and alias-aware typestate analysis framework named PATA, to effectively detect OS bugs. We have evaluated PATA on the Linux kernel and three popular IoT OSes to detect three common types of bugs (null-pointer dereferences, uninitialized-variable accesses and memory leaks). We also experimentally compare PATA to seven state-of-the-art static analysis approaches, and PATA finds many real bugs missed by these approaches. In the evaluation, PATA in total finds 574 real bugs with a low false positive rate of 28%, and 206 of these real bugs have been confirmed by OS developers.

ACKNOWLEDGMENT

We thank our shepherd, Yuanyuan Zhou, and anonymous reviewers for their helpful advice on the paper. We also thank OS developers, who gave useful feedback and advice to us. This work was supported by the National Natural Science Foundation of China under Project 62002195 and Australian Research Grants DP210101348. Jia-Ju Bai is the corresponding author.

REFERENCES

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Cophenhagen.

[2] Marcelo Arroyo, Francisco Chiotta, and Francisco Bavera. 2016. An user configurable clang static analyzer taint checker. In *Proceedings of the 35th International Conference of the Chilean Computer Science Society (SCCC)*. 1–12. https://doi.org/10.1109/SCCC.2016.7835996.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th International Conference on Programming Language Design and Implementation (PLDI)*. 259–269. https://doi.org/10.1145/2666356.2594299.

[4] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 255–268.

[5] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. 2020. Effective detection of sleep-in-atomic-context bugs in the Linux kernel. *ACM Transactions on Computer Systems (TOCS)* 36, 4 (2020), 1–30. https://doi.org/10.1145/3381990.

[6] Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu. 2016. Testing error handling code in device drivers using characteristic fault injection. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. 635–647.

[7] George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. 2017. A datalog model of must-alias analysis. In *Proceedings of the 6th International Workshop on State Of the Art in Program Analysis*. 7–12. https://doi.org/10.1145/3088515.3088517.

[8] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. 2010. SLAM2: static driver verification with under 4% false alarms. In *Proceedings of the 2010 International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 35–42.

[9] Mohamad Barbar and Yulei Sui. 2021. Compacting points-to sets through object clustering. In *Proceedings of the 2021 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 1–27. https://doi.org/10.1145/3485547.

[10] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2020. Flow-sensitive type-based heap cloning. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP 2020)*. 24:1–24:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.24.

[11] Petr Bauch, Vojtěch Havel, and Jiří Barnat. 2014. LTL model checking of LLVM bitcode with symbolic data. In *Proceedings of the 2014 International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS)*. Springer, 47–59. https://doi.org/10.1007/978-3-319-14896-0_5.

[12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *Proceedings of the 8th International Symposium on Operating Systems Design and Implementation (OSDI)*. 209–224.

[13] Ben-Chung Cheng and Wen-Mei W Hwu. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI)*. 57–69. https://doi.org/10.1145/349299.349311.

[14] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*. 480–491. https://doi.org/10.1145/1250734.1250789.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 265–278. https://doi.org/10.1145/1961296.1950396.

[16] Clang 2021. Clang: an LLVM-based C/C++ compiler. http://clang.llvm.org/.

[17] CLOC 2021. CLOC: count lines of code. https://cloc.sourceforge.net.

[18] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. 2015. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. 361–372. https://doi.org/10.1145/2771783.2771811.

[19] Kai Cong, Fei Xie, and Li Lei. 2013. Symbolic execution of virtual devices. In *Proceedings of the 13th International Conference on Quality Software*. 1–10. https://doi.org/10.1109/QSIC.2013.44.

[20] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: interface aware fuzzing for kernel drivers. In *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)*. 2123–2138. https://doi.org/10.1145/3133956.3134069.

[21] Coverity 2021. Coverity: a commercial static analysis tool. https://scan.coverity.com/.

[22] Coverity reports for Linux 2021. Coverity reports for Linux kernel. https://github.com/torvalds/linux/search?q=coverity&type=commits.

[23] Coverity reports for Zephyr 2021. Coverity reports for Zephyr project. https://github.com/zephyrproject-rtos/zephyr/labels/Coverity.

[24] Cppcheck 2021. Cppcheck: a tool for static C/C++ code analysis. http://cppcheck.sourceforge.net/.

[25] CSA 2021. Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[26] Manuvir Das. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI)*. 35–46. https://doi.org/10.1145/358438.349309.

[27] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the 23rd International Conference on Programming Language Design and Implementation (PLDI)*. 57–68. https://doi.org/10.1145/512529.512538.

[28] Alain Deutsch. 1994. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the 15th International Conference on Programming Language Design and Implementation (PLDI)*. 230–241. https://doi.org/10.1145/773473.178263.

[29] Dinakar Dhurjati, Manuvir Das, and Yue Yang. 2006. Path-sensitive dataflow analysis with iterative refinement. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*. 425–442. https://doi.org/10.1007/11823230_27.

[30] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th International Symposium on Operating System Design (OSDI)*. 1–16.

[31] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 72–82. https://doi.org/10.1109/ICSE.2019.00025.

[32] Stephen Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA)*. 133–144. https://doi.org/10.1145/1348250.1348255.

[33] Bolei Guo, Neil Vachharajani, and David I August. 2007. Shape analysis with inductive recursion synthesis. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*. 256–265. https://doi.org/10.1145/1250734.1250764.

[34] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. A system and language for building system-specific, static analyses. In *Proceedings of the 23rd International Conference on Programming Language Design and Implementation (PLDI)*. 69–82. https://doi.org/10.1145/512529.512539.

[35] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th International Symposium on Principles of Programming Languages (POPL)*. 226–238. https://doi.org/10.1145/1594834.1480911.

[36] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 2011 International Symposium on Code Generation and Optimization (CGO)*. 289–298. https://doi.org/10.1109/CGO.2011.5764696.

[37] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation (PLDI)*. 254–263.

https://doi.org/10.1145/381694.378855.
[38] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 4 (1999), 848–894. https://doi.org/10.1145/325478.325519.
[39] Infer 2021. Facebook Infer: a tool to detect bugs in Java and C/C++/Objective-C code. https://fbinfer.com/.
[40] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. 1998. Single and loving it: must-alias analysis for higher-order languages. In *Proceedings of the 25th International Symposium on Principles of Programming Languages (POPL)*. 329–341. https://doi.org/10.1145/268946.268973.
[41] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. 2019. Fuzzing error handling code in device drivers based on software fault injection. In *Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE)*. 128–138. https://doi.org/10.1109/ISSRE.2019.00022.
[42] Vineet Kahlon. 2008. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*. 249–259. https://doi.org/10.1145/1379022.1375613.
[43] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. 2018. An efficient data structure for must-alias analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC)*. 48–58. https://doi.org/10.1145/3178372.3179519.
[44] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th International Symposium on Operating Systems Principles (SOSP)*. 147–161. https://doi.org/10.1145/3341301.3359662.
[45] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. 2010. Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology (IST)* 52, 2 (2010), 210–219. https://doi.org/10.1016/j.infsof.2009.10.004.
[46] Goh Kondoh and Tamiya Onodera. 2008. Finding bugs in Java native interface programs. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*. 109–118. https://doi.org/10.1145/1390630.1390645.
[47] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. 2010. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*. 1–14.
[48] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*. 278–289. https://doi.org/10.1145/1273442.1250766.
[49] Yuxiang Lei and Yulei Sui. 2019. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *Proceedings of the 26th International Static Analysis Symposium (SAS)*. 27–47. https://doi.org/10.1007/978-3-030-32304-2_3.
[50] LLVM 2021. LLVM compiler infrastructure. https://llvm.org/.
[51] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th International Conference on Computer and Communications Security (CCS)*. 1867–1881. https://doi.org/10.1145/3319535.3354244.
[52] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium*. 1769–1786.
[53] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *Proceedings of the 2012 International Conference on Verified Software: Tools, Theories, Experiments (VSTTE)*. Springer, 146–161. https://doi.org/10.1007/978-3-642-27705-4_12.
[54] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2004. Precise call graphs for C programs with function pointers. *Automated Software Engineering* 11, 1 (2004), 7–26. https://doi.org/10.1023/B:AUSE.0000008666.56394.a1.
[55] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*. 247–260. https://doi.org/10.1145/1357010.1352618.
[56] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: don't interpret, compile!. In *Proceedings of the 30th USENIX Security Symposium*. 181–198.
[57] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: correctness checking for real code. In *Proceedings of the 24th USENIX Security Symposium*. 49–64.
[58] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. 2012. SymDrive: testing drivers without devices. In *Proceedings of the 10th International Symposium on Operating Systems Design and Implementation (OSDI)*. 279–292.
[59] RIOT 2021. RIOT: a real-time multi-threading operating system. https://github.com/RIOT-OS/RIOT.
[60] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium*. 167–182.
[61] Semantic patches of Coccinelle 2021. Project to study faults in Linux. https://github.com/coccinelle/faults-in-Linux.
[62] V Shakti D Shekar, BB Meshram, and MP Varshapriya. 2012. Device driver fault simulation using KEDR. *International Journal of Advanced Research in Computer Engineering and Technology* (2012), 580–584.
[63] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the extensional scalability problem for value-flow analysis frameworks. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 812–823. https://doi.org/10.1145/3377811.3380346.
[64] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th International Conference on Programming Language Design and Implementation (PLDI)*. 693–706. https://doi.org/10.1145/3192366.3192418.
[65] Smatch 2021. Smatch: a static bug-finding tool for C. http://smatch.sourceforge.net/.
[66] Robert E Strom and Shaula Yemini. 1986. Typestate: a programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)* 1, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929.
[67] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*. 160–170. https://doi.org/10.1145/2854038.2854043.
[68] Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue. 2018. Loop-oriented pointer analysis for automatic simd vectorization. *ACM Transactions on Embedded Computing Systems (TECS)* 17, 2 (2018), 1–31. https://doi.org/10.1145/3168364.
[69] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering (TSE)* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311.
[70] SVF wiki 2021. SVF wiki. https://github.com/SVF-tools/SVF/wiki/Detecting-memory-leaks.
[71] Syzkaller 2021. Syzkaller: a kernel fuzzer. https://github.com/google/syzkaller.
[72] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. 2021. Undo workarounds for kernel bugs. In *Proceedings of the 30th USENIX Security Symposium*.
[73] TencentOS-tiny 2021. TencentOS-tiny: a real-time IoT operating system developed by Tencent. https://github.com/Tencent/TencentOS-tiny.
[74] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 999–1010. https://doi.org/10.1145/3377811.3380386.
[75] Wenwen Wang. 2021. MLEE: effective detection of memory leaks on early-exit paths in OS kernels. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*. 31–45.
[76] WLLVM 2021. WLLVM: whole program LLVM. https://github.com/travitch/whole-program-llvm.
[77] Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. 2014. ARC++: effective typestate and lifetime dependency analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. 116–126. https://doi.org/10.1145/2610384.2610395.
[78] Zhiwu Xu, Dongxiao Fan, and Shengchao Qin. 2016. State-Taint Analysis for Detecting Resource Bugs. In *Proceedings of the 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 168–175. https://doi.org/10.1016/j.scico.2017.06.010.
[79] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*. 155–165. https://doi.org/10.1145/2001420.2001440.
[80] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*. 42–54. https://doi.org/10.1145/3134600.3134620.
[81] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*. 144–154. https://doi.org/10.1145/2338965.2336771.
[82] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-based selective flow-sensitive pointer analysis. In *Proceedings of the 21st International Static Analysis Symposium (SAS)*. 319–336. https://doi.org/10.1007/978-3-319-10936-7_20.
[83] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization (CGO)*. 218–229. https://doi.org/10.1145/1772954.1772985.
[84] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: sanitizing API usages through semantic cross-checking. In *Proceedings of the 25th USENIX Security Symposium*. 363–378.
[85] Z3 2021. Z3: a theorem prover. https://github.com/Z3Prover/z3.

[86] Zephyr 2021. Zephyr: a scalable real-time operating system. https://github.com/zephyrproject-rtos/zephyr.

[87] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *Proceedings of the 28th International Symposium on the Foundations of Software Engineering (FSE)*. 221–232. https://doi.org/10.1145/3368089.3409686.

[88] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 829–845. https://doi.org/10.1145/2660193.2660213.

[89] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th International Symposium on Principles of Programming Languages (POPL)*. 197–208. https://doi.org/10.1145/1328438.1328464.