Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels

Hang Zhang
hzhan033@ucr.edu
UC Riverside

Weiteng Chen
wchen130@ucr.edu
UC Riverside

Yu Hao
yhao016@ucr.edu
UC Riverside

Guoren Li
gli076@ucr.edu
UC Riverside

Yizhuo Zhai
yzhai003@ucr.edu
UC Riverside

Xiaochen Zou
xzou017@ucr.edu
UC Riverside

Zhiyun Qian
zhiyunq@cs.ucr.edu
UC Riverside

ABSTRACT

Static analysis is known to yield numerous false alarms when used in bug finding, especially for complex vulnerabilities in large code bases like the Linux kernel. One important class of such complex vulnerabilities is what we call "high-order taint style vulnerability", where the taint flow from the user input to the vulnerable site crosses the boundary of a single entry function invocation (*i.e.,* syscall). Due to the large scope and high precision requirement, few have attempted to solve the problem.

In this paper, we present SUTURE, a highly precise and scalable static analysis tool capable of discovering high-order vulnerabilities in OS kernels. SUTURE employs a novel summary-based high-order taint flow construction approach to efficiently enumerate the cross-entry taint flows, while incorporating multiple innovative enhancements on analysis precision that are unseen in existing tools, resulting in a highly precise inter-procedural flow-, context-, field-, index-, and opportunistically path-sensitive static taint analysis.

We apply SUTURE to discover high-order taint vulnerabilities in multiple Android kernels from mainstream vendors (*e.g.,* Google, Samsung, Huawei), the results show that SUTURE can both confirm known high-order vulnerabilities and uncover new ones. So far, SUTURE generates 79 true positive warning groups, of which 19 have been confirmed by the vendors, including a high severity vulnerability rated by Google. SUTURE also achieves a reasonable false positive rate (51.23%) perceived by users of our tool.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; • **Theory of computation** → **Program reasoning**.

KEYWORDS

OS kernels; Vulnerability discovery; Static program analysis

ACM Reference Format:
Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. 2021. Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3484798

Hang Zhang is a postdoc researcher at Georgia Tech at the time of publication.
The second to sixth authors contribute equally in helping with the evaluation.

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.
© 2021 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484798

1 INTRODUCTION

A major weakness of static analysis for bug finding is the high false positive rate, which is one reason why dynamic approaches such as fuzzing is gaining much more popularity where any bugs found are technically true positives. This weakness is especially significant with large and complex software such as the Linux kernel, where many bugs are triggered after a sequence of syscall invocations. Specifically, the large state space created by multiple program entry points (permutation of syscalls) makes it extremely challenging for static analysis to be both precise and scalable. On the other hand, a fuzzer like Syzkaller [3] can generate random but meaningful test cases involving different sequences of syscalls guided by coverage, and proves to be effective at finding such bugs. However, there is no guarantee that it will be able to uncover all bugs due to its nature of random exploration.

One important class of the aforementioned complex bugs is the *high-order taint style vulnerabilities*, where an attacker-controlled input (*i.e.,* taint source) is propagated to sensitive operations (*i.e.,* taint sink) without proper sanitization, following a complex control/data flow involving multiple entry function invocations. For example, an entry function `A()` copies its user-provided argument to a global variable `G`, which is later used as an array index unchecked in another entry function `B()`, causing an out-of-bound access. The *order* here refers to the number of entry function invocations that are needed to trigger the vulnerability. Compared to the simple "one-shot" taint vulnerabilities where the taint propagation is confined within a single entry function invocation (*i.e.,* first-order), high-order bugs frequently seen in the stateful software (*e.g.,* Linux kernel) are much more difficult to uncover, due to the need to reason about the complicated cross-entry taint propagation.

Ideally, we want a static analysis tool that can systematically analyze the program to identify the high-order vulnerabilities with a good coverage, while minimizing false alarms. However, this is a difficult task because of the following specific challenges:

Challenge 1. The tool needs to efficiently enumerate cross-entry taint flows. Intuitively, since multiple entry functions can be invoked in any order, an analysis needs to walk through many possible permutations and repeatedly analyze a same entry in different permutations, which is a significant scalability challenge.

Challenge 2. The analysis needs to be accurate and precise enough to handle the cross-entry taint flows which can be lengthy. Since these flows are usually "concatenated" from multiple local flows within individual entry functions, any inaccuracy will accumulate and eventually cause an unacceptable number of false alarms.

Given these challenges, although there are many existing works on statically discovering taint style vulnerabilities [8, 10, 26, 40], few can discover high-order ones. The closest work is by Dahse *et al.* [14], focusing on only the second-order vulnerability in web applications, which are very different and relatively simple compared to Linux kernel (*e.g.,* higher-level programming languages compared to C, fewer entry points and fixed taint propagation paths around the limited central data storage).

In this paper, we develop a novel static analysis tool that can address the above challenges and discover high-order (arbitrary orders) taint vulnerabilities in the Linux kernel (and potentially any other stateful C programs) effectively and efficiently. To overcome the first challenge (*i.e.,* scalability), our core idea is to first analyze each entry function independently (only for once) and create an abstract summary regarding its taint behaviors for both local and global variables, then in the vulnerability discovery phase, we construct high-order taint flows on demand by querying the individual summaries. This enables an efficient high-order taint flow enumeration. As for the second challenge (*i.e.,* accuracy and precision), we integrate many innovative and/or practical features into the static analysis to boost its precision. They include an opportunistic path-sensitive analysis piggybacked through a flow-sensitive one, handling ambiguous all-to-all memory updates, and many others that result in a highly precise inter-procedure flow-, context-, field-, index-, and opportunistic path-sensitive static taint analysis. Besides, we also make considerable efforts handling kernel code patterns (*e.g.,* indirect calls).

We evaluate our tool on driver modules of different Android kernels used in various mobile devices (*e.g.,* Google, Samsung, Huawei). The results show that our tool can discover previously unknown high-order taint vulnerabilities. So far, our tool has reported 79 true positive warning groups, of which 19 have been confirmed by developers, including one high severity vulnerability as rated by Google. Our tool also achieves a reasonable false positive rate as perceived by the warning reviewers (51.23%) and an acceptable performance (*e.g.,* concurrently analyze all 37 modules of a target kernel within 30 hrs).

We summarize our major contributions as below:

(1) To our best knowledge, we are the first to attempt to systematically and statically discover high-order taint style vulnerabilities in the Linux kernel. Our method can also be easily generalized to other stateful software.

(2) We implement a prototype tool SUTURE, which is able to construct high-order taint flows with high-precision points-to and taint analyses, making it general enough for our problem as well as others requiring static taint analysis. We will open source SUTURE[1] to facilitate the reproduction of results and future research.

(3) We successfully discover previously unknown high-order taint vulnerabilities in the kernel and report them to the developers, including high-severity ones.

[1] https://github.com/seclab-ucr/SUTURE

```
00 struct data { int32_t a; char b[4]; } d;
01
02 entry0(int cmd, char user_input) {
03   switch(cmd) {
04   case 0:
05     d.b[0] = user_input; break;
06   default:
07     foo(cmd,user_input);
08   }
09 }
10
11 foo(int n, char c) {
12   if (n == 0)
13     d.b[1] = c;
14 }
```

```
15 entry1() {
16   bar((char*)&d);
17   ...
18   d.b[0] = 0;
19 }
20
21 bar(char *p) {
22   *(p+4) += 0xf0;  // (1) ⚠
23 }
24
25 entry2() {
26   char a[8];
27   a[0] = d.b[1] + 0xf0;  // (2) ⚠
28   ...
29 }
```

Local Taint Flows:		Calling Sequences:
entry0:	*entry2:*	*entry0* → *entry1* : Overflow
user_input ▶ d.b[0]	d.b[1] ▶ (2)⚠	*entry0* → *entry2* : ✓
entry1:	d.b[1] ▶ a[0]	*entry1* → *entry1* : ✓
d.b[0] ▶ (1)⚠		......

* Red: Input directly provided by the user, Blue: Global variables.

Figure 1: An Example of High-Order Bug Discovery

2 OVERVIEW

In this section we describe the architecture and overall workflow of SUTURE, with a motivating example.

2.1 The Motivating Example

Fig. 1 shows an abstracted example of high-order vulnerabilities. There are three entry functions, *i.e.,* `entry0()`, `entry1()` and `entry2()`, that can be invoked in any order. First, we note a byte overflow at line 22 (site `(1)`). To trigger it, however, one needs to first invoke `entry0()` with `cmd=0`, so that the user provided `user_input` flows to the global variable `d.b[0]` (line 5). Then, a different entry function `entry1()` needs to be invoked, which subsequently invokes `bar()`. At that point, the value of `d.b[0]` (previously set by the user input in `entry0()` whose address is aliased with `p+4`) is retrieved and involved in an overflow-inducing addition operation (line 22).

The most important characteristic of this vulnerability is that the taint flow from the original user input to the final overflow site is relayed via a global variable, making it a high-order (more specifically, second-order) taint style vulnerability. As mentioned in §1, it is difficult for existing tools to statically discover such vulnerabilities since they usually only reason about the local taint flows within a single entry function. For example, from the scope of only `entry1()`, we do not know `d.b[0]` is actually controlled by the user, while blindly assuming it can cause excessive false alarms.

Constructing high-order taint flows can be challenging, because analyzing all possible permutations of entry invocations is not scalable. Moreover, any analysis imprecision can be amplified when stitching results from individual entry functions. We illustrate a few potential sources of imprecision below.

(1) A path-insensitive analysis will wrongly conclude that `entry0()`, with a non-zero `cmd`, can propagate `user_input` to `d.b[1]` in its callee `foo()` (line 13), which is impossible since the conflicting path conditions at line 6 and line 12. Such path-insensitivity can eventually create a false alarm that line 27 in `entry2()` (when invoked after `entry0()`) can cause an overflow.

(2) To discover the potential overflow at line 22, the static analysis must be able to accurately resolve the pointer arithmetic and figure out that `*(p+4)` is an alias to `d.b[0]`, otherwise a false negative will occur. Additionally, the analysis also needs to be inter-procedural in order to figure out that argument `p` of `bar` aliases to `d`;
(3) An index-insensitive analysis may issue a false alarm at line 27, following the calling sequence `entry0(cmd=0,...) -> entry2()`, because `d.b[0]` and `d.b[1]` are not differentiated. If combined with path-insensitivity in *(2)*, there will likely be many more false alarms after line 27 under the calling sequence `entry0(cmd=1,...) -> entry2`, since the whole array `a` can be over-tainted instead of only `a[0]`. Moreover, since `d` contains `d.b` as an embedded array, the analysis also needs to handle such nested structures;
(4) The analysis must also correctly recognize that line 18 in `entry1()` kills the existing taint of `d.b[0]`, so that additional `entry1()` invocations after the first one will no longer trigger the overflow at line 22. This requires a cross-entry flow-sensitivity.

Unfortunately, to our best knowledge, no existing static analysis tools possess all these precision we desire (with more manifested later in §3). This motivates SUTURE, a highly precise static analysis tool capable of discovering high-order taint vulnerabilities.

2.2 Workflow

We provide an overview of SUTURE in this section. The architecture of SUTURE is shown in Fig. 2. We briefly describe its workflow in four stages:
1. Input. SUTURE requires as input the LLVM bitcode file compiled from the target program, along with a config file which specifies the entry function information (*e.g.,* function names, user-controlled arguments). For the motivating example in Fig. 1, `entry0()`, `entry1()`, and `entry2()` will be listed as three entry functions in the config file, where the `user_input` argument of `entry0()` is specified as the user controllable input.
2. Static Taint Analysis. SUTURE will then perform a precise static taint analysis and generate an independent summary for each entry function, which includes all the local taint flows within it. In this phase both user input and global variables are treated as taint source, while the local taint flow to every variable used in the entry function is recorded. In Fig. 1 we show the local taint flows of the three entry functions in the bottom left box (we omit the intermediate variables visible only at the bitcode level for site `(1)` and `(2)`.). It is worth noting that SUTURE analyzes each entry function only once in an order-insensitive way, *e.g.,* `entry0()` can be analyzed either before or after `entry1()`), avoiding the expensive cost of repeatedly analyzing a same entry in different calling sequences, however, SUTURE can still discover high-order vulnerabilities as detailed later.
3. Vulnerability Discovery. In this stage, SUTURE tries to stitch together various entry functions with various vulnerability detectors to pinpoint different types of bug-inducing program statements (*e.g.,* an arithmetic operation may cause integer overflow). For each such statement, SUTURE decides whether any cross-entry taint flows exist from user input to the problematic statement. In the motivating example, the local taint flows of `entry0()` and `entry1()` are stitched to form the high-order taint flow of the vulnerability.

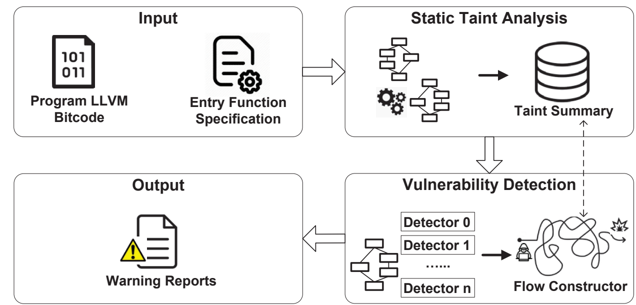

Figure 2: System Architecture of SUTURE

4. Output. For each issued warning, SUTURE outputs any relevant information such as the warning type and full cross-entry taint flow, SUTURE also calculates the order for each warning. For the motivating example, SUTURE eventually fires one valid warning whose calling sequence is shown in the right bottom (a second-order vulnerability), while avoiding all the false alarms as mentioned in §2.1.

3 STATIC ANALYSIS DESIGN

In this section we describe the design of SUTURE, including the various enhancements made to make the precise and efficient high-order taint analysis possible.

3.1 Positioning

Given the LLVM bitcode files and the entry function specification, the goal of static taint analysis is to construct a taint summary (detailed in §3.3) for each entry function, to achieve this goal, we independently analyze each entry function and record its taint facts. Our static taint analysis follows the basic design and reuses the main data structures of Dr. Checker [26], which makes a soundy inter-procedure traversal of each entry function in the top-down style and for each visited LLVM IR, performing the alias and taint analysis, which updates the points-to and taint information associated with variables involved in the IR. The rules for points-to record update and taint propagation are quite standard as manifested in [26], so we will not elaborate on them again. Basically, Dr. Checker's static analysis is context-, flow-, and field- sensitive. However, as detailed later, all these sensitivity are partial or limited (see §3.6.1 and §3.6.3) which needs to be addressed in SUTURE. Moreover, SUTURE also has many additional requirements for the static analysis compared to Dr. Checker. Throughout the section, we primarily focus on describing our enhancements over it.

In this section, we will first describe three novel features of SUTURE that are not found in other static analysis tools, including the essential techniques to support scalable high-order taint flow construction (§3.3) and several innovative techniques to improve analysis precision and efficiency (§3.4 and §3.5). Then we describe various other improvements in SUTURE (§3.6) that are although mostly well-known, but rarely packed together to achieve a highly precise static analysis, which is critical for high-order bug discovery since mistakes can be amplified when multiple local taint flows connect (§2.1).

3.2 Definitions

Before we delve into the details, we start with a set of definitions to simplify the later description of the design.

Def 0 An entry function ε of a program module (*e.g.,* a kernel module) serves as a part of the module interface, thus it does not have any callers within the same module and intends to be invoked directly by the user or other modules (*e.g.,* top-level `ioctl()` functions of a driver).

Def 1 The taint source $\mathbb{S}$ includes both user-provided arguments of entry functions ($\mathbb{U}$) and all globally accessible variables or memory regions ($\mathbb{G}$ which we refer to as global memory). Note that $\mathbb{G}$ contains both explicitly defined global variables (*e.g.,* a global integer) and the ones reachable from them, *e.g.,* a global object containing a pointer field pointing to heap memory. This can go across arbitrary layers of pointer indirection. Formally:

$$\mathbb{S} = \mathbb{U} \cup \mathbb{G}$$
$$\mathbb{U} = \{v|\exists\varepsilon, v \text{ is a user argument of } \varepsilon\}$$
$$\mathbb{G} = \{v|v \text{ is globally accessible}\}$$

Def 2 A calling context $\triangle$ is defined as a sequence of instructions:

$$\triangle = [i_0, i_1, ..., i_{2n}]$$

An instruction with an even subscript denotes the entry instruction of a caller function, while the odd denotes a call site instruction within the caller (*e.g.,* i_2 is the entry of the function that is called at i_1), the sequence always ends with the entry instruction of current executing function, so its length is always odd. This definition enables us to differentiate multiple callees at a same call site (*e.g.,* an indirect call with multiple potential targets).

Def 3 We define an "instruction location" (*InstLoc* for short in the remaining paper) as an instruction i plus the calling context $\triangle$ (**Def 2**) it is executed in, we use I to denote an *InstLoc* to differentiate it with a static instruction i:

$$I = (i, \triangle)$$

Def 4 A taint flow τ is basically an *InstLoc* sequence:

$$\tau = [I_0, I_1, ..., I_n]$$

The first *InstLoc* I_0 initiates the taint propagation from one taint source variable $v \in \mathbb{S}$, while the remaining *InstLoc*s pass the taint on.

Def 5 We define the connect operator $\circ$ for taint flows as following:

$$\tau_0 = [I_{00}, I_{01}, ..., I_{0n}], \tau_1 = [I_{10}, I_{11}, ..., I_{1n}]$$
$$\tau_0 \circ \tau_1 = \begin{cases} [I_{00}, ..., I_{0n}, I_{10}, ..., I_{1n}], & \text{if sink}(I_{0n})\text{==src}(I_{10}) \\ \varnothing, & \text{else} \end{cases}$$

This basically says that two taint flows can be sequentially connected *iff* the last *InstLoc* of one taint flow propagates the taint to a variable that is used as the taint initiator at the beginning of the other taint flow.

Def 6 We now define the "order" of a taint flow with the $order()$ function, before that, we need to first define the $reach()$ function to test the reachability between two *InstLoc*s:

$$reach(I_0, I_1) = \begin{cases} True, \text{if } \exists\varepsilon, I_0 \text{ can reach } I_1 \text{ on } ICFG(\varepsilon) \\ False, \text{else} \end{cases}$$

$ICFG(\varepsilon)$ means the inter-procedure control flow graph of the entry function ε, I_0 can reach I_1 on $ICFG(\varepsilon)$ implies that there is at least one execution of ε that can reach the *InstLoc* I_0 and after that, I_1. With the $reach()$ definition:

$$order(\tau) = |\{k|I_k \in \tau, I_{k+1} \in \tau, \neg reach(I_k, I_{k+1})\}|$$

Intuitively, $order(\tau)$ is the number of "break points" in τ, where to continue following the taint flow we have to make another entry function invocation. We hereby call a taint flow τ **high-order taint flow** if $order(\tau) > 1$.

It is worth noting that since $reach()$ is not transitive (*e.g.,* $reach(I_0, I_1) \land reach(I_1, I_2) \nrightarrow reach(I_0, I_2)$, because the path between I_0 and I_1 may pose conflicting constraints to that between I_1 and I_2), our definition of $order()$ can underestimate the real taint flow order. In other words, there can be more "break points" in a taint flow than counted by $order()$ (*e.g.,* $reach(I_0, I_1) \land reach(I_1, I_2)$ results in no "break points" between I_0 and I_2 by $order()$, but there could be one.). However, it may be expensive to calculate the real order due to the need of constraint solving. Besides, we find that the underestimation rarely happens in practice.

Def 7 We define the local taint flow set of an entry function ε as LT_ε. It is the set of all taint flows that can be produced within one invocation of ε, naturally, we have $\forall\tau \in LT_\varepsilon, order(\tau) == 1$.

3.3 Summary-Based High-Order Taint Flow Construction

The foremost challenge SUTURE needs to address is to efficiently construct high-order taint flows in the face of the enormous space of the possible calling sequences of entry functions. To avoid repeatedly analyzing a ε in different sequences as a naive solution might do, SUTURE employs a summary based method, where each ε only needs to be analyzed once for summary generation, then SUTURE can efficiently construct high-order taint flows, by connecting the local taint flows as mentioned in **Def 5**. Note that the global variable acts as waypoints in connecting the local taint flows, *e.g.,* one local flow may propagate user input taint source to a global variable and then another local flow may propagate the same global variable (as source) to a critical sink. We detail the two main steps of this process below.

3.3.1 Taint Summary Generation. The taint summary of an entry function ε is basically its local taint flow set LT_ε (**Def 7** in §3.2), in other words, the summary records all local taint flows originating from $\mathbb{S}$ (**Def 1** in §3.2) and sinking to every accessed variable (local or global) within $ICFG(\varepsilon)$. SUTURE organizes the local taint flows in LT_ε by the sink variables - each sink variable is associated with a set of local taint flows (τ) reaching it, while the source variable can be obtained from the taint tag (§3.6.5) associated with each τ. This enables a quick query of τ by sink, as well as the connect operation (**Def 5** in §3.2) for constructing high-order taint flows.

Conceptually, SUTURE's taint summary is similar to those used in prior bottom-up static analysis work [9, 10, 40]. However, one important difference is that SUTURE relies on the summaries to connect multiple top-level entry function invocations, instead of connecting a caller to a callee (*e.g.,* by applying the callee's summary at the call site). As such, SUTURE has a special focus on the shared

typedef struct {int X;} foo; struct {foo *p;} G; foo F;	

e0(u0) {
G.p = malloc(...);
G.p->X = u0;
}
In Summary: G.p -> obj0 (solid)
T0 : u0 ▶ obj0.X
	e1() {
	G.p = &F;
	int a = G.p->X + 1;
	}
	In Summary: G.p -> F (solid)
	T1 : F.X ▶ a

e2(u2) {
int a = G.p->X + 1;
G.p->X = u2;
}
In Summary: G.p -> obj2 (dummy)
T2 : obj2.X ▶ a; T3 : u2 ▶ obj2.X
	e3(u3) {
	int a = G.p->X + 1;
	G.p->X = u3;
	}
	In Summary: G.p -> obj3 (dummy)
	T4 : obj3.X ▶ a; T5 : u3 ▶ obj3.X

Figure 3: Examples of Implicit Global Memory Matching

states in the taint summary, it comprehensively models the global memory (see **Def 1** in §3.2) of arbitrary layers of pointer indirection. Specifically, whenever pointers are involved in global variables, it can be challenging to resolve them because the memory they point to can in theory be changed in any ε. It is therefore tricky to reason about such global pointers, which represents a unique challenge for connecting local taint flows of top-level entry functions.

3.3.2 High-Order Taint Flow Construction. Connecting two local taint flows is relatively straightforward. However, to do so correctly, we discuss two important considerations below.

Global Memory Matching. As mentioned earlier, two local taint flows can be connected only when one's sink matches the other's source (**Def 5** in §3.2). Since high-order taint flows are relayed via global memory, the question becomes how to match the global memory tainted in one flow with the one used in another flow. As mentioned in **Def 1**, SUTURE handles two types of global memory: explicitly defined and those reachable by global pointers. For the former, we can simply match them by their identifiers. However, things get more complicated for the pointer case.

Consider the example in Fig. 3, `e0()` and `e1()` each assigns `G.p` (a pointer field in a global object) to either a dynamically allocated heap object or statically defined one, while both `e2()` and `e3()` directly access whatever object pointed to by `G.p`. In this situation SUTURE must correctly "guess" the relationship between the objects visited in the four entry functions in order to connect the local taint flows. For example, `e0()` and `e1()` obviously visit the different object instances, so although `u0` flows to `G.p->X` in `e0()`, and the seemingly same `G.p->X` flows to `a` in `e1()`, there is no way for `u0` to flow into `a`, because *obj0* in τ_0 (the heap object) cannot match *F* in τ_1 (statically defined). However, since `e2()` and `e3()` can access either *obj0* or *F* (depends on whether `e0()` or `e1()` is called earlier), we should allow their local τ to be connected to each other, or to those in `e0()` and `e1()`. For example, τ_3 can connect τ_1 because *obj2* can potentially be *F*. Similarly, τ_3 can also connect τ_4 because *obj2* and *obj3* can be identical).

To summarize, when objects are accessed but not defined in a ε (*e.g., obj2* and *obj3*), we do not necessarily know which objects are used (*e.g.,* can be either *obj0* or *F*), so we create a placeholder dummy object. Instead, such bindings are instantiated by access-path matching (*e.g.,* both *obj2* and *F* can be accessed via `G.p`) when connecting local taint flows.

Taint Overwrite. Another point worth discussing is that not all τ should be preserved for flow connection. For example, a global memory may be tainted during the middle of a ε invocation but later untainted. Similarly, it may be tainted by different sources at different points in the function. Therefore, SUTURE filters out those τ whose taint will be overwritten later. Note that this may prevent SUTURE from discovering some taint-style concurrency bugs, where an intermediate taint of a global memory in one ε can be visible to another. In favor of limiting the false taint flows, we leave a better treatment of concurrency situations as future work.

3.4 Opportunistic Path Sensitivity

It is well known that static analysis can follow infeasible paths due to unawareness of conflicting path constraints, causing both inaccuracy (*e.g.,* impossible taint propagation) and inefficiency (*e.g.,* analyzing unnecessary branches). The straightforward solution is to adopt path-sensitivity, however, a fully path-sensitive analysis can be overly expensive, due to complex constraint solving and path explosion. We thus aim to utilize path-sensitivity whenever possible, while avoiding having to pay the high cost. To this end, we propose what we call opportunistic path-sensitive analysis. We make the design based on two important observations:

(1) A good fraction of the path constraints in the kernel are simple, and yet collecting and solving them would allow us to prune a large number of infeasible paths.

(2) It is possible to piggyback some form of path-sensitive analysis into the workflow of a flow-sensitive analysis.

Based on the above, our idea is to opportunistically collect path constraints during the flow-sensitive analysis and only in the following simple forms: $v\ op\ C$, where v is a variable, C is a constant (*e.g.,* a literal number), and $op \in \{==, >, <, \geq, \leq\}$. Specifically, whenever our flow-sensitive analysis enters a conditional branch, we collect the corresponding constraint if it is in such a simple form. Whenever branches merge, we remove the constraints. At a first glance, no path-sensitive analysis is allowed if we piggyback the flow-sensitive analysis in this way, since at the merge point we lost the constraints for individual branches.

However, we note that within one branch, it is possible that additional conditional statements can occur (intra- or inter-procedure), making it possible for us to trim infeasible paths with the opportunistic path-sensitivity. We show a real world example in Fig. 4. As we can see, `msm_lsm_ioctl()` calls `msm_lsm_ioctl_shared()` at line 3, under one specific switch case with the `cmd` restricted to `SNDRV_LSM_REG_SND_MODEL_V2`, the same `cmd` is passed to the callee and used again as the switch conditional at line 8, since its value has already been restricted at the call site (line 3), there is actually only one valid switch case in the callee (line 9) under this calling context. Our opportunistic path-sensitive analysis can collect the equality constraint on `cmd` at line 3 and propagate it to the callee. This allows us to filter out 16 out of 17 infeasible switch cases due to the conflicting constraints, simultaneously improving accuracy and efficiency.

3.5 Multi-Source Multi-Sink Pairing

One unique challenge for static analysis we identified during our study is the multi-source multi-sink pairing problem. If not

```
00 static int msm_lsm_ioctl(..., unsigned int cmd, ...) {
01   switch (cmd) {
02   case SNDRV_LSM_REG_SND_MODEL_V2:
03     msm_lsm_ioctl_shared(..., cmd, ...); break;
04   case ...
05   }
06 }
07 static int msm_lsm_ioctl_shared(..., unsigned int cmd, ...) {
08   switch (cmd) {
09   case SNDRV_LSM_REG_SND_MODEL_V2: ......; break;
10   case ...                         //17 cases in total
11   }
12 }
```

Figure 4: An Example of Opportunistic Path-Sensitivity

properly handled, explosion of points-to records and taints can happen, leading to a massive number of false positives. Fig. 5 illustrates the problem with a concrete example. When starting to analyze the function `start_endpoints()`, the argument subs is a pointer that points to two instances of `snd_usb_substream` according to the previous static analysis results. Consequently, the left side of the assignment at line 2 can be one of two memory locations (*i.e.,* `data_subs` field of `snd_usb_endpoint`, either instance 0 or 1), while the right side `subs` points to either instance 0 or 1 of `snd_usb_substream`. In this situation, the common way to perform the assignment (as used in many popular static analysis tools like Dr. Checker [26] and SVF [34]) is all to all (*e.g.,* `data_subs` field of `snd_usb_endpoint 0` will point to both `snd_usb_substream` 0 and 1). However, it is obvious that in the real program execution, `data_subs` can only point back to its own parent `snd_usb_substream` instance (*e.g.,* 0 to 0 and 1 to 1). We call this multi-source multi-sink pairing problem, failure to pair the two sides (*e.g.,* all-to-all update) will create many superfluous data flow facts.

To solve this problem, our key observation is that in the aforementioned scenario, two sides of the assignment actually share the same source of multiplicity (*e.g.,* the left side `ep0->data_subs` at line 2 has two possible locations because `ep0` can point to two structure instances, which is again because `subs` is so at line 1, the same reason for the right side), thus, as long as the unique source "collapses" to one of many possibilities in the runtime, both sides of the assignment "collapse" as well. Following this observation, for every LLVM IR that can serve as a "source of multiplicity", *e.g.,* a *phi* instruction can aggregate multiple points-to/taint records from different paths to its receiver variable, SUTURE assigns each individual outcome record a unique label $< IR, i >$ (i is a numeric value to differentiate multiple outcome records of the multiplicity IR), which will also be propagated to all derived records. For example, in Fig. 5 the pointer `ep0` is derived from `subs`, and the latter's two points-to records for `snd_usb_substream 0` and 1 have their labels respectively inherited by `ep0`'s two records for `snd_usb_endpoint` 0 and 1. By matching these labels, when the multi-to-multi assignment happens (*e.g.,* line 2) we can precisely pair the source and the sink if they share the same source of multiplicity, bringing the $2 * 2$ update to two $1 * 1$ ones in Fig. 5.

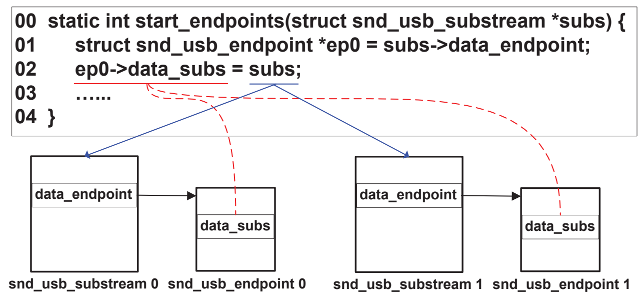

Figure 5: An Example of Multi-Source Multi-Sink Pairing

```
00 struct data {              if.then:
01   int a,b; } d;              %add = add 1, %c
02                              %0 = load i32* GEP (%struct.data* @d, 0, 0)
03 foo(int c) {                 %add1 = add %0, %add
04   int r = 1;                 store %add1, i32* GEP (%struct.data* @d, 0, 0)
05   if (c>0) {                 br label %if.end
06     r += c;
07     d.a += r;              if.else:
08   } else {                   %sub = sub 1, %c
09     r -= c;                  %1 = load i32* GEP (%struct.data* @d, 0, 0)
10     d.a -= r;                %sub2 = sub %1, %sub
11   }                          store %sub2, i32* GEP (%struct.data* @d, 0, 0)
12 }                            br label %if.end
```

Figure 6: Necessity of Memory SSA based Analysis

3.6 Other Improvements in SUTURE

3.6.1 Memory SSA based Analysis. One major source of inaccuracy of Dr. Checker (and many other LLVM based static analysis) is the lack of memory SSA (Static Single Assignment) form [11]. While the top-level variables in LLVM IR are inherently put in the SSA form [28], the address-taken memory objects are not, causing difficulties when implementing flow- and context-sensitive analysis. For example, in Fig. 6, two redefinitions of the same local variable `r` (line 6 and 9) results in two individual top-level variables at the LLVM IR level (*i.e.,* `%add` and `%sub`), so the static analysis can easily associate the later usage with the unique definition simply by the LLVM variable identifier (*e.g.,* `%add`). Based on this built-in SSA form, Dr. Checker achieves the flow- and context- sensitivity for the top-level LLVM variables. However, multiple redefinitions of the address-taken memory object field (*e.g.,* `d.a` at line 7 and 10) do not result in different memory cells, instead, they go to the same memory location (*e.g.,* the two `store` in Fig. 6), in other words, the SSA form for memory objects is not enforced in LLVM IR. In this situation, the static analysis must be able to correctly correlate the `load` of a memory cell to one of (potentially) many `store`, otherwise, it will lost the flow- and context- sensitivity for the widespread address-taken memory objects, causing both false positives and negatives (*e.g.,* over- and under- taint), as happened in Dr. Checker.

To address this problem, SUTURE implements an on-the-fly memory SSA analysis. Specifically, we append an $InstLoc$ to each points-to/taint update of a memory cell to represent where the update happens (*e.g.,* one `store` instruction in Fig. 6, together with

the calling context of `foo()`). Based on such information, SUTURE can correctly figure out which points-to/taint records should be propagated to a certain use site of the same memory cell, by performing an inter-procedure reachability test between the update site and the use site. The reachability test is implemented based on the topology and the dominance relationship of the control flow graph (*e.g.,* if a strong points-to update site post-dominates a previous one on a same memory cell, the old points-to will be masked out from the new site), since the algorithm is standard, we omit its details here.

3.6.2 Index Sensitivity. Besides being field-sensitive, SUTURE is also index-sensitive (*i.e.,* the ability to differentiate individual array elements), whose importance has already been shown in the motivating example in §2.1. In principle, our design of index-sensitivity follows two rules for array read/write respectively:

(1) If an array element is read with a constant index (*e.g.,* v = a[2]), we return the points-to/taint records related to exactly that index; If the index is a variable (*e.g.,* v = a[i]), we conservatively merge the records of all array elements and return them.

(2) If an array element is written with a constant index (*e.g.,* a[2] = v), we perform a strong update (*i.e.,* the new records can overwrite the old ones) for exactly that index; If the index is a variable (*e.g.,* a[i] = v), we conservatively update every array element, and the update is weak (*i.e.,* new records co-exist with old ones).

3.6.3 General Language Feature Support. In this section we discuss our enhancements in SUTURE for two C language features that are critical for analysis accuracy.

Nested Structure. Nested structure (*i.e.,* one structure is embedded as a field in a parent structure) is a widely used language feature and failure to correctly handle it can significantly impact the field-sensitivity, for example, Dr. Checker only differentiates the top-layer fields in the parent structure but not those in the embedded ones, which can cause issues like over-tainting.

SUTURE addresses this problem by recursively creating a new abstract memory object for each embedded field when it is accessed, while maintaining the relationship between the new object and the parent object, this way, SUTURE supports nested structure of arbitrary layers. We also carefully design the LLVM IR processing logics in the points-to and taint analysis to take nested structure into account, for example, SUTURE processes all indices of the *GEP* instruction instead of only the first 2 since it is required for accessing the fields within the embedded structures.

Pointer Arithmetic. It is well known that pointer arithmetic in the C language family can often cause inaccuracies in static analysis, since it is difficult to keep track of the exact pointer location during the arithmetic calculation, *e.g.,* normally, LLVM IR accesses the *2nd* field of a structure by simply specifying the field number *2* in the *GEP* instruction with the structure base pointer, however, in some cases (*e.g.,* optimization) the field can be accessed by directly subtracting an offset (between the *2nd* and *5th* fields) from the pointer to the *5th* field. To handle such cases, SUTURE records the detailed layout (*e.g.,* size and offset of each field in bytes) of each structure and faithfully calculates the new target field after pointer arithmetic according to the pointer type (*e.g.,* pointer conversion aware, like `(char*)p-1` and `(int32*)p-1` are different), offset to add/sub, and the structure layout. It is worth noting that our pointer arithmetic handling has a byte-level accuracy and we always try to align to the field boundary, though rare, this may cause some inaccuracies (*e.g.,* a pointer to the middle of a field, or bit-level pointer arithmetic), we leave the handling of these cases as the future work.

Detector	Description
ITDUD	Tainted data usage in risky functions, *e.g.,* `strcpy()`
TAD	Tainted arithmetic operations, *e.g.,* integer overflow
TLBD	Tainted loop bound conditions, *e.g.,* infinite loops
TPDD	Tainted pointer dereference, *e.g.,* arbitrary mem write

Table 1: Vulnerability Detectors used in SUTURE

3.6.4 Kernel Code Pattern Handling. To better support the analysis of the kernel code, SUTURE also takes care of some special kernel code patterns, of which the most important one is the prevalent indirect calls. Dr. Checker uses the type-based method to resolve indirect call targets, though being a common and standard solution, it can cause many false positives in practice. To further improve the accuracy, SUTURE employs a method similar to PeX [42], which takes advantages of domain knowledge on the kernel coding paradigm and resolves the indirect call targets by matching the parent structure and field id of the function pointer.

3.6.5 Multi-Tag Taint Analysis. To construct high-order taint flows we must be able to differentiate multiple taint sources (see **Def 1,5** in §3.2), *e.g.,* in Fig. 1 we must know exactly that local taint flow of `entry1()` originates from the specific global variable `d.b[0]` to connect it to that of `entry0()`. So instead of only maintaining the binary "tainted or not" state, SUTURE associates a unique taint tag for $\forall v \in \mathbb{S}$, the tag will also be propagated to all the tainted variables by the related source, enabling us to easily query the taint sources for each τ.

4 VULNERABILITY DISCOVERY AND WARNING GROUPING

After generating the taint summary for each ε, SUTURE can then proceed to discover the high-order taint vulnerabilities and output the warning report. This process includes two steps: (1) identify the instructions that can potentially trigger vulnerabilities (*e.g.,* an arithmetic instruction may cause an integer overflow), this is done by various vulnerability detectors. For each identified instruction, SUTURE confirms the existence of the vulnerability by deciding whether any involved variable is tainted (can be through a high-order flow) by the user, and (2) fire and group warnings for confirmed vulnerabilities. In this section, we detail the important aspects of these two steps.

Vulnerability Detectors. Dr. Checker has a collection of simple but well-defined vulnerability detectors, each targets instructions of a certain pattern (*e.g.,* conditional jump at the loop bound). Since our goal is to discover taint style vulnerabilities, we reuse Dr. Checker's 4 detectors aiming at them. We list our selected detectors and a brief description of their purposes in Table 1, more details can be found in the original paper [26]. We leave the development of more

detectors as future work. As mentioned above, SUTURE's novelty mainly lies in its ability to construct high-order taint flows, which is independent of the detectors.

On-Demand Query of Taint Summaries. One way to apply the vulnerability detector is to first construct the high-order taint summaries by considering all permutations of entry functions, which can be overly expensive. Instead, we construct the high-order taint flows on demand in a backward fashion. In other words, SUTURE provides a query utility ("Flow Constructor" in Fig. 2) which takes a sink variable as input and taint flows from $\mathbb{U}$ to it (including high-order ones) as output. For instance, we first apply the vulnerability detector on each individual entry function by looking at its local taint flows. If a warning is generated (*e.g.,* a potential integer overflow happens due to an addition operation taking a tainted operand), we check if the taint source is user input or some global memory. If it is global memory, we need to query the summaries of other entry functions with a matching sink (same global memory being the sink). There, if the corresponding source is an user input, we conclude that it is a second-order warning. We make this design choice because it is more flexible (*e.g.,* enabling us to focus only on sink variables of interest) and fits better into the workflows of many existing static bug finding tools [8, 26], which first pinpoints potential vulnerable sites in the program with various detectors, and then take a closer look at the involved variables (*e.g.,* decide whether it is tainted by user).

It is worth noting that although SUTURE is able to discover vulnerabilities of arbitrary orders by recursively performing the aforementioned backward query, we observe that in practice it is highly unlikely to have true high-order vulnerabilities above the order of 4 (most likely false positives if it is longer than that). In fact, our evaluation shows most true positives are second-order. Therefore, we will stop searching for higher-than-4 order taint flows when the current query already takes too long a time.

Warning Grouping. If an user tainted variable is detected in a sensitive instruction (defined by the vulnerability detector as mentioned in §4), a warning will be fired for it. SUTURE assembles all the warnings issued in the input program as its output. Each warning specifies the warned instruction and its calling context, the warning type (*e.g.,* integer overflow), and the complete taint flow(s) from the user input to the sink site. SUTURE also calculates the *order* of each taint flow and attaches it to each warning.

One important observation we have during the warning review process is that many warnings often share a same "prefix" in the initial taint propagation while are only slightly different in the final warning sites or taint sinks. For example, a is the tainted variable in an overflow inducing instruction $c = a+b$, and c, which is tainted by a, is then immediately used as a loop bound, in this situation, two raw warnings will be generated for the overflow and the loop bound respectively, obviously, instead of reviewing the two warnings one by one, a better way is to first inspect the common trace prefix from the original user input to a, and if no problem, then the different sinks (often close to each other).

Based on this observation, to help the reviewers screen the warnings more efficiently, SUTURE groups the similar warnings together from the data flow perspective, regardless of the warning types as listed in Table 1 (*i.e.,* warnings of different types can be put in a same group.). More specifically, two warnings will be grouped together if i) their warning sites (*i.e.,* the warned instruction) locate in a same function f, and ii) their taint propagation traces share a same sub-trace starting from the entry of f. With this grouping strategy, the reviewer can avoid studying the shared taint trace over and over and quickly go through a warning group by only carefully inspecting a small subset, greatly reducing the required review time. A real-world example of warning grouping can be found in §6.7.

5 IMPLEMENTATION

As mentioned before, SUTURE is built on top of Dr. Checker, however, to improve the accuracy of the static analysis and support high-order taint flow construction, we re-write most parts of Dr. Checker's static analysis and implement many new functionalities (detailed in §3), in total, compared to Dr. Checker, SUTURE has 14,482 LOC added and 2,741 LOC removed in C++, plus 630 LOC of python scripts. In this section we discuss some implementation details of SUTURE.

LLVM Version. Dr. Checker is based on LLVM 3.8, which is too old to compile newer kernel versions nowadays. To test the latest kernels, SUTURE is based on LLVM 9.0.

Driver Module and Entry Function Identification. We follow Dr. Checker's approach [26] to identify the vendor driver modules and their entry functions. However, we make some improvements including (1) besides the modules identified by keyword search in the kernel config file, we also review the kernel source tree to include any missing ones, and (2) we update some out-of-date kernel structure definitions in the Dr. Checker's entry identification script, as well as include some missing `ioctl()` functions.

False Alarms Filtering. We use some simple but reliable heuristics to filter out certain obvious false alarms. Specifically, we cut off the taint flows through (1) a modulo operation if the modulus is a small integer (current threshold is 64), and (2) a logical "and" operation if only limited number of bits (current threshold is 6) are not cleared. Basically, these situations suggest that the tainted value is a well bounded "index" or "flag" over which the user has a very limited control, thus unlikely to cause security issues. We leave a more systematical and principle false alarms filtering as a future work, as will be discussed in §7.

6 EVALUATION

In this section we show the evaluation results of SUTURE as both a static analysis engine (*e.g.,* the efficacy of our static analysis improvements in §3) and a high-order bug finding tool (*e.g.,* regarding its accuracy, efficiency, and bug finding ability).

6.1 Experiment Setup and Procedure

Dataset. We evaluate SUTURE on the driver modules extracted from a diverse set of kernels used in flagship Android devices, manufactured by different vendors and based on different chipsets, we summarize them in Table 2. The last column in the table lists the number of driver modules we extract and test for the corresponding kernel. Besides the relatively new kernels in the table, we also compile older versions of the Qualcomm kernel modules which contain 4 known high-order taint vulnerabilities (as seen in Table 5), to test whether SUTURE can successfully catch them.

No.	Vendor	Chipset	Model	Version	#Modules
0	Google	Qualcomm	Pixel 4 XL	4.14.150	37
1	Samsung	Exynos	Galaxy S20	4.19.87	20
2	Huawei	Hisilicon	Mate 40 Pro	4.14.116	30
3	Xiaomi	Mediatek	Redmi K30 Ultra	4.14.141	29

Table 2: Tested Android Kernels

Hardware Configuration. We run the evaluation on a server with Intel Xeon E5-2695 v4 CPU @ 2.10GHz and 256 GB RAM.
Procedure. For each selected kernel in Table 2, we first try to extract its vendor-specific driver modules and then identify their entry functions, as described in §5. With the above input we run SUTURE for high-order taint vulnerability discovery, the output warning groups (§4) are then manually reviewed by us to decide true and false positives. It is worth noting that although SUTURE is also capable of discovering the simple first-order taint vulnerabilities (*i.e.,* only require one entry function invocation), in the evaluation we focus on the high-order ones only.

6.2 Efficacy of SUTURE's Static Analysis Improvements

To achieve a highly precise analysis SUTURE ships with many different static analysis improvements as detailed in §3. To better understand whether and how they benefit the analysis precision and efficiency, we randomly pick 20 modules from the Qualcomm kernel and run an instrumented SUTURE on them, collecting statistics during the process for each of SUTURE's improvements. These statistics can be categorized into three groups as shown in Table 3:
(1) For opportunistic path-sensitivity (**PATH**), we can see that it helps SUTURE get rid of infeasible basic blocks and/or callees (a callee can contain many more basic blocks that we do not count in Table 3) in 19 out of 20 modules, boosting both performance and accuracy. For example, we find the analysis time of one large module decreases from 54 hrs to 31 hrs by applying the opportunistic path-sensitivity.
(2) For multi-source multi-sink pairing (**MSMS**), memory-SSA (**MEMSSA**), and index-sensitivity (**INDEX**), our statistics show that they can effectively trim false positive points-to records and taint flows. Specifically, besides the data in Table 3, **MEMSSA** also captures missing points-to records by Dr. Checker in all 20 modules (Min/Med/Max: 10/44/1351), though Dr. Checker adopts an always weak taint update policy (*e.g.,* no overwriting for old taint records), missing points-to records will inevitably lead to false negative taint flows from the very beginning. It is worth noting that both false positive and negative data flow facts will accumulate even more (possibly exponentially) if left unrecognized as the analysis proceeds, thus, our statistics here are significantly underestimated.
(3) For pointer arithmetic (**POINTER**) and nested structure (**NEST**), we observe that in all tested modules, there is a considerable subset of GEP operations (*i.e.,* the main LLVM IR responsible for calculating the structure field offset) that need to handle them - otherwise, both false positive and negative analysis errors can happen and accumulate.

Improvement	#Affected Modules	#Infeasible BBs (Min/Med/Max)	#Infeasible Callees (Min/Med/Max)
PATH (§3.4)	19	9/153/115336	0/8/2414
		#Reduced Points-To (Min/Med/Max)	#Reduced Taint Flows (Min/Med/Max)
MSMS (§3.5)	18	8/174/852492	15/886/1024094
MEMSSA (§3.6.1)	20	10/1789/547733	52/5139/2480710
INDEX (§3.6.2)	20	0/14/17850	18/2922/276318
		Ratio of Affected GEP Operations (Min/Med/Max)	
POINTER (§3.6.3)	20	3%/13%/79%	
NEST (§3.6.3)	20	12%/27%/37%	

Table 3: Statistics on SUTURE's Static Analysis Improvements from 20 Randomly Selected Qualcomm Modules

Kernel No.	#Warning Groups					$\#TP^2$	$\#FP_r^3$ (R^4)
	$ITDUD^0$	TAD^0	$TLBD^0$	$TPDD^0$	$Unified^1$		
0	0	188	87	277	488	30	22 (42.31%)
1	0	137	41	147	281	17	12 (41.38%)
2	0	201	63	171	365	22	22 (50.00%)
3	0	240	62	280	469	10	27 (72.97%)
SUM	0	766	253	875	1603	79	83 (51.23%)

0: #groups of warnings issued by specific detectors (Table 1) only.
1: #groups by standard grouping strategy regardless of warning types (§4);
2: #groups manually confirmed by us as true positives.
3: #false alarm groups as **perceived by the reviewers**. (§6.3)
4: **Reviewer perceived** false positive rate: $\#FP_r/(\#FP_r+\#TP)$ (§6.3)

Table 4: Vulnerability Discovery Accuracy of SUTURE

We further build a baseline version of Dr. Checker augmented only with multi-tag taint analysis (§3.6.5) that is essential for taint flow connection, but not any other enhancements, to verify whether it can identify the same high-order vulnerabilities as discovered by SUTURE. The result shows that none of our 19 vendor confirmed warnings are identified (*i.e.,* for each high-order warning at least one component local taint flow is not recognized), because i) the analysis is stuck due to too many false positive points-to and taint records (*e.g.,* the analysis progress of one module is almost frozen after 329 hrs, around which point each LLVM variable has tens of thousands of points-to and taint records on average), ii) failure to resolve correct indirect call targets (§3.6.4), and iii) missing taint propagation due to failure to handle pointer arithmetic and nested structure. These results well justify the necessity and efficacy of SUTURE's various static analysis improvements in §3.

6.3 Vulnerability Discovery Accuracy

We show the evaluation results regarding the vulnerability discovery accuracy of SUTURE in Table 4. SUTURE in total fires 1,603 high-order warning groups, where 79 are confirmed as true positives by our manual inspection. Furthermore, 19 out of the 79 have been confirmed by the corresponding vendors. At the first glance, this results in a very high false positive rate of 95.07% ((1603-79)/1603) which seems completely unusable in practice. However, this is far from the truth. In the remaining of this section,

```
00 int mtk_session_set_mode(..., unsigned int session_mode) {
01    ...
02    if (session_mode >= MTK_DRM_SESSION_NUM) {
03          goto error;
04    }
05    ...
06    mtk_crtc_path_switch(...,mode_tb[session_mode].ddp_mode[i], 1);
07    ...
08 }
```

Figure 7: A Taint Trace Segment Leading to False Alarms

CVE	Bug Type	Severity[1]	Order[2]	Discovered
CVE-2016-2068	Integer Overflow	High	2	Yes
CVE-2016-5859	Integer Overflow	High	2	Yes
CVE-2017-0608	Buffer Overflow	High	2	Yes
N/A[3]	Buffer Overread	N/A[3]	2	Yes

1: Based on the CVSS score in the CVE entry. 2: Def 6 in §3.2.
3: We cannot locate a CVE number, the patch can be found in [4].

Table 5: Evaluation on Known High-Order Vulnerabilities

we will first describe the root causes behind these false alarms, and then explain why the actually perceived false positive rate by the SUTURE users is much lower (*i.e.,* 51.23%).

False Positive Analysis. We summarize the major causes of SUTURE's false alarms as following.

(1) Ignored Constraints for Tainted Variables. Except the simple false alarm filtering heuristics described in §7, SUTURE in general conservatively keeps all the taint flows, without reasoning about the constraints posed on the tainted variables, which can lead to false positive warnings (*e.g.,* the tainted variable in the vulnerable site has been properly sanitized). We show a concrete example in Fig. 7. The argument `session_mode` of `mtk_session_set_mode()` is user-controllable, which is then used to index the array `mode_tb` at line 6. SUTURE thus determines that the whole retrieved array element `mode_tb[session_mode]` is tainted, this is true since the user does have the choice on which element to access. However, `session_mode` is upper bound checked at line 2 and the array `mode_tb` is also predefined, so the user cannot really control the content of the obtained array element on desire. As a result, continuing the taint propagation from `mode_tb[session_mode]` causes false alarms subsequently. So far, the ignored constraints is the most common FP cause for SUTURE, as well as many other static analysis based bug detection works.

(2) Recursive Data Structures. The recursive data structure (*e.g.,* linked lists) is another well-known difficulty for static analysis, since it is hard to statically differentiate their contained elements. To be conservative, SUTURE does not differentiate the elements in the linked list which is widely-used in kernel. Though being a common practice, it can cause false alarms in many cases, for an example, a local taint flow τ_0 sinks to element 0 of a linked list while τ_1 sources from a different element 1, in theory, τ_0 cannot be connected to τ_1 since $sink(\tau_0) \neq source(\tau_1)$ (**Def 5** in §3.2), but SUTURE connects them due to element-insensitivity, resulting in invalid high-order taint flows.

(3) Infeasible Paths. By nature, static analysis may follow infeasible paths, leading to false alarms. There are two reasons for SUTURE to encounter with infeasible paths: i) SUTURE recovers infeasible indirect call targets, and ii) SUTURE fails to recognize conflicting path constraints with its opportunistic path-sensitivity (§3.4) because they are too complicated. This FP cause is actually the least common out of the three.

As can be seen from the above analysis, *the core problem behind the false alarms is **not** that SUTURE generates inaccurate local taint flows, on the contrary, in almost all cases, these local flows are valid and precise (*e.g., *no over-taint)*, demonstrating SUTURE's value as a highly precise static taint analysis for a single entry function. In other words, bug finding tools go beyond the requirement of precise taint flow tracking - they also need to rigorously reason about the constraints (reason *(1)* and *(3)*) where SUTURE falls short.

Other than the above, SUTURE's false alarm count is greatly boosted by high-order taint flow construction. First, even if two local taint flows are both valid, connecting them incorrectly can cause FPs (*e.g.,* reason *(2)*), secondly, any false positives encountered in one local taint flow will be "multiplied" by its connection to potentially many other local flows. However, this also means that many false alarms share exactly the same problematic sub- taint trace. Exploiting this fact, the actual false positive rate perceived by reviewers is orders of magnitude smaller with the following review procedure: i) the reviewer picks and inspects a warning, if it is a FP, then also identifies the problematic sub-trace which is basically an instruction sequence in string format; ii) automatically filter out all other warnings containing the same sub-trace by string match; iii) pick the next warning to review from the filtered pool. As a concrete example, once recognized, the taint trace in Fig. 7 helps us immediately exclude 94 warning groups without additional reviewing efforts.

We hence define the **reviewer perceived false positive rate** R as $R = \#FP_r/(\#FP_r + \#TP)$, where $\#FP_r$ is the number of false alarms that actually need the reviewer to carefully inspect one by one (*e.g.,* does not include automatically filtered out ones) and the $\#TP$ is the count of valid warning groups. R represents SUTURE's false positive rate in the real-world review scenario, as shown in Table 4, SUTURE achieves an aggregated R of 51.23%, which is much more acceptable for a static analysis tool. We will discuss potential ways to further reduce the false positive rate in §7.

6.4 Known High-Order Taint Vulnerabilities

It is usually difficult to test the false negative rate of a bug finding tool due to the lack of ground truth, this is especially true for SUTURE since there is no available large dataset of high-order taint bugs. Thus, as a small-scale validation, we assemble four known high-order taint vulnerabilities and confirm that SUTURE can successfully re-discover them in older versions of driver modules, as shown in Table 5. Though not being a comprehensive evaluation, we can still envision potential reasons for false negatives.

False Negative Analysis. We summarize some potential false negative causes as following.

(1) Soundy Analysis. SUTURE is built on top of Dr. Checker and inherits its soundy but not sound static analysis (*e.g.,* skip the general kernel functions and limit the loop iteration times) for efficiency and accuracy [26], which can possibly lead to false negatives. Moreover, since the general kernel functions are

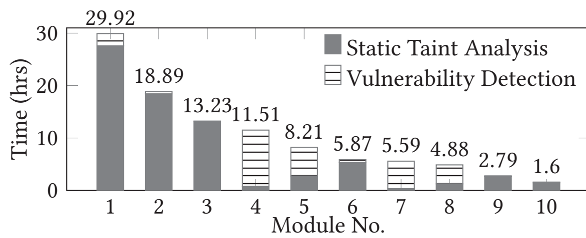

Figure 8: Time Breakdown for Qualcomm Modules (> 1 hr)

skipped, Dr. Checker and SUTURE need to model the behaviors of some important utility functions such as `copy_from_user()` (*e.g.*, as a taint source), in this situation, it is possible for SUTURE to miss some vulnerabilities due to the incomplete or incorrect modeling. In fact, this has already happened in Dr. Checker due to the lack of modeling of `memdup_user()` (another taint initiator function), which we add in SUTURE.

(2) Incomplete Call Chains. SUTURE cannot guarantee to analyze all possible call chains, mainly due to two reasons: i) SUTURE may miss indirect call targets, and ii) SUTURE limits the call stack depth when analyzing each entry function in the top-down manner for scalability, currently, the depth limit is 8, increased from 5 in Dr. Checker.

We will discuss possible fixes to some of these issues in §7.

6.5 Efficiency

SUTURE analyzes one kernel module with one instance on a single CPU core, so naturally, multiple modules can be concurrently analyzed on multi-core systems that are widely available nowadays (*e.g.*, our evaluation server has 72 cores). Thus, the efficiency bottleneck is mainly those most demanding modules. In Fig. 8, we show the time cost of all Qualcomm kernel modules which take more than 1 hr to analyze. The time cost covers two parts: i) the static analysis to generate per-entry taint summaries (§3), and ii) the vulnerability discovery involving on-demand taint flow construction (§4). Basically, SUTURE's static analysis is significantly more costly than Dr. Checker's - even the most costly module in latter's evaluation takes about only 16 mins [26]. This performance, however, is well expected due to our improvements on static analysis precision (§3). The vulnerability discovery time is generally not correlated to the static analysis time, because it mainly relies on the amount of cross-entry shared states - the more of them, potentially the more high-order taint flows we need to construct during the vulnerability discovery process.

Due to space limit, we only show the performance details of the Qualcomm kernel as the other kernels bear a similar characteristic. As a conclusion, we believe SUTURE achieves a reasonable efficiency (*e.g.*, analyzes all Qualcomm modules concurrently within 30 hrs) given its precision and capability.

6.6 Discussion of *Order*

One interesting aspect of the warnings generated by SUTURE is the *order* (**Def 6** in §3.2), denoting the count of required entry function invocations to trigger the vulnerability. Though SUTURE by design is capable of discovering vulnerabilities of arbitrary order, so far, most of our confirmed true positive warnings in the evaluation are second-order, SUTURE generates one valid third-order warning, however, the same vulnerability can also be reproduced by a simplified second-order taint flow. That being said, we create some artificial benchmark programs that contain vulnerabilities triggerable by higher-order taint flows only (up to ninth-order) and confirm that SUTURE can successfully pinpoint them. Based on these results, we envision that SUTURE can discover higher-order vulnerabilities with an extended scope (*e.g.*, more kernel modules and user-space programs).

6.7 Study of Discovered Vulnerabilities

In this section we discuss the vendor-confirmed high-order vulnerabilities discovered by SUTURE so far. By root cause, the 19 confirmed warning groups can be further categorized into 6 major issues (*e.g.*, under the same root cause there can be different warnings for different calling contexts or instructions). We have 1 issue rated as high severity and 3 as medium by the vendors, which can cause arbitrary memory read/write or privilege escalation. For the remaining 2, the developers confirmed that they can cause out-of-bound memory access, but with unclear security impact (*e.g.*, the over-read data may not contain sensitive information), thus need further investigation. We are still in the process of reporting and waiting for confirmation of other discovered issues. By nature, the high-order taint vulnerabilities have security impacts as severe as the well-known first-order ones, however, they are more stealthy (and thus dangerous) due to the complicated control and data flows, making SUTURE a valuable tool. We will discuss SUTURE's potential in discovering more security vulnerabilities in §7. To better illustrate the discovered high-order issues, in the remaining section we study two representative cases in detail.

Case 1. To trigger the vulnerability in Fig. 9 we need two steps (*i.e.*, a second-order vulnerability). First, a user needs to invoke the entry function `snd_ctl_ioctl()`, which eventually reaches `msm_pcm_put_out_chs()` that propagates a user input (in red) to a global variable `channel_mixer[fe_id].output_channel` (in blue) at line 3. Then, after `snd_ctl_ioctl()` returns, another entry function `snd_pcm_ioctl()` needs to be invoked. Following its callchain, `channel_mixer+fe_id`, which is equivalent to &channel_mixer[fe_id], is passed as an argument to a callee `adm_programable_channel_mixer()` at line 9, the corresponding formal argument, `ch_mixer`, is then used in the overflow inducing operation at line 17, to access `ch_mixer->output_channel` that aliases to `channel_mixer[fe_id].output_channel`, which is the global variable controllable by the user in the first step (line 3). Consequently, the user can overflow `param_size` at line 17 or `sz` at line 22, since the latter is used as an allocation size (line 23), the allocated buffer can be much smaller than expected due to the overflow, causing out-of-bound access later.

It is worth noting that, besides the high-order nature, this vulnerability also involves indirect calls (`snd_ctl_elem_write()` -> `msm_pcm_put_out_chs()` in step 1), pointer arithmetic (line 10), and nested structure (line 3), making it difficult to be statically discovered. We also want to mention that SUTURE will group warnings for the potential overflows at line 17 and 22 together as

```
STEP 1:
CALLCHAIN: snd_ctl_ioctl -> snd_ctl_elem_write_user ->
snd_ctl_elem_write -> msm_pcm_put_out_chs

00 static int msm_pcm_put_out_chs(struct snd_kcontrol *kcontrol,
01                        struct snd_ctl_elem_value *ucontrol) {
02   ...
03   channel_mixer[fe_id].output_channel =
04              (unsigned int)(ucontrol->value.integer.value[0]);
05   return 1;
06 }

STEP 2:
CALLCHAIN: snd_pcm_ioctl -> ... -> msm_pcm_routing_channel_mixer ->
adm_programable_channel_mixer

07 static int msm_pcm_routing_channel_mixer(int fe_id, ...) {
08   ...
09   ret = adm_programable_channel_mixer(..., ..., ..., ...,
10                                       channel_mixer + fe_id, ...);
11   ...
12 }
13
14 int adm_programable_channel_mixer(..., ..., ..., ...,
15                          struct msm_pcm_channel_mixer *ch_mixer, ...) {
16   ...
17   param_size = 2 * (4 + ch_mixer->output_channel +
18              ch_mixer->input_channels[channel_index] +
19              ch_mixer->input_channels[channel_index] *
20              ch_mixer->output_channel); //potential overflow ⚠
21   ...
22   sz = ... + param_size; //potential overflow ⚠
23   adm_params = kzalloc(sz, GFP_KERNEL);
24   ...
25 }
```

Figure 9: Case Study 1: A High-Order Vulnerability Discovered by SUTURE

described in §4, leading to a more natural and easier review process for the auditors.

Case 2. As shown in Fig. 10, we again need two entry function invocations to trigger the vulnerability. First, `snd_ctl_ioctl()` is invoked with proper arguments so that `iaxxx_put_pdm_bclk()` can be subsequently called, within which the user can set the value of a global variable `iaxxx->pdm_bclk` (line 5). Then, unlike the first case study, we invoke the same entry function `snd_ctl_ioctl()` again but with different arguments, so that this time we follow a different call chain to eventually reach `iaxxx_pdm_port_setup()`, which uses the same global variable `iaxxx->pdm_bclk` unchecked to index a fixed-length array `pdm_cfg`, causing out-of-bound accesses (line 13-15). This issue has been rated as high severity by Google.

This case has some notable characteristics. First, even if there is only one entry function (*e.g.,* `snd_ctl_ioctl()`), high-order taint analysis is still necessary as some callees are mutually exclusive within one entry invocation (*e.g.,* `iaxxx_put_pdm_bclk()` and `iaxxx_pdm_port_setup()`). Second, it is actually not so straightforward as it appears to match the global structure `iaxxx` used at line 5 and 12, because both `iaxxx` are local instead of explicitly defined global variables (see line 4 and 10). In this situation, SUTURE must rely on its implicit global memory matching (covered in §3.3.2) to decide the identity of the two occurrences (*e.g.,* both `iaxxx` can be reached from a shared `kcontrol` object following a same access path).

```
STEP 1:
CALLCHAIN: snd_ctl_ioctl -> snd_ctl_elem_write_user ->
snd_ctl_elem_write -> iaxxx_put_pdm_bclk

00 static int iaxxx_put_pdm_bclk(struct snd_kcontrol *kcontrol,
01                              struct snd_ctl_elem_value *ucontrol)
02 {
03   struct snd_soc_codec *codec = snd_soc_kcontrol_codec(kcontrol);
04   struct iaxxx_codec_priv *iaxxx = dev_get_drvdata(codec->dev);
05   iaxxx->pdm_bclk = ucontrol->value.enumerated.item[0];
06   return 0;
07 }

STEP 2:
CALLCHAIN: snd_ctl_ioctl -> snd_ctl_elem_write_user ->
snd_ctl_elem_write -> iaxxx_pdm_portb_put -> iaxxx_pdm_port_setup

08 static int iaxxx_pdm_port_setup(...) {
09   struct snd_soc_codec *codec = snd_soc_kcontrol_codec(kcontrol);
10   struct iaxxx_codec_priv *iaxxx = dev_get_drvdata(codec->dev);
11   ...
12   pdm_bclk = iaxxx->pdm_bclk;
13   port_sample_rate = pdm_cfg[pdm_bclk].sample_rate; //OOB ⚠
14   words_per_frm = pdm_cfg[pdm_bclk].words_per_frame; //OOB ⚠
15   word_len = pdm_cfg[pdm_bclk].word_length; //OOB ⚠
16   ...
17 }
```

Figure 10: Case Study 2: A High-Order Vulnerability Discovered by SUTURE

7 LIMITATIONS AND DISCUSSIONS

In this section we summarize SUTURE's limitations and discuss potential improvements in the future.

Soundness. As mentioned in §6.4, SUTURE's static analysis is not sound (*e.g.,* no fixed-point loop analysis, limited call stack depth). Although we intentionally sacrifice the soundness for a better performance and accuracy similar as in Dr. Checker [26], it can lead to false negatives regarding vulnerability discovery. One possible way to improve soundness is to adopt the bottom-up style static analysis [9, 10, 40] (*i.e.,* callees are analyzed and summarized before callers) when constructing the taint summary for each entry function, which alleviates the limitations on call stack depth due to the improved efficiency (*e.g.,* a same callee will not be repeatedly analyzed). We leave this as a future work.

Recursive Data Structure Handling. SUTURE does not differentiate the elements in recursive data structures (*e.g.,* linked lists) to (1) be conservative, and (2) simplify the access path to ease the global object matching (§3.3.2) involving recursive structures (*e.g.,* there can be numerous access paths from one linked list element to another). However, this design choice also contributes significantly to false alarms as mentioned in §6.3. To better handle the recursive data structures and suppress the false positives, we envision a more fine-grained static analysis which can differentiate the accessed elements (*e.g.,* by considering the conditions used to select a specific element, such as comparing the element id against a desired value), or the integration of a dynamic analysis which can help verify whether two element access are the same utilizing the runtime information.

Path Constraints Reasoning. Besides the simple path constraints considered in opportunistic path-sensitivity (§3.4) to filter out infeasible paths, SUTURE does not reason about the path

constrains associated with feasible taint flows, resulting in the major body of false alarms as mentioned in §6.3. There are existing solutions for this problem in previous works, Sys [8] and UBITECT [40] employ limited-scale symbolic execution to validate the discovered vulnerabilities without introducing high performance penalties, KINT [36] carefully reasons about the constraints specifically for integer overflow vulnerabilities. Though not the focus of this paper, we believe these approaches can be naturally combined with SUTURE to further reduce false alarms, which we leave as a future work.

Vulnerability Scope. To demonstrate the efficacy of SUTURE's high-order analysis capability, in this paper we choose to reuse several of Dr. Checker's detectors for discovering high-order taint vulnerabilities in the kernel (§4). However, the techniques packed in SUTURE can also be applied to discover a wider range of vulnerabilities in a broader set of software, for example, many use-after-free vulnerabilities (*e.g.,* [1, 2]) have a cross-entry nature (*e.g.,* the "free" happens in one entry invocation while the "use" happens in another), where SUTURE's cross-entry data flow analysis can be useful. Moreover, SUTURE can also scan general C programs other than the kernel.

8 RELATED WORK

Statically Discovering Taint Vulnerabilities. There is a large amount of work trying to statically discover taint style vulnerabilities, for different software written in different programming languages and at different layers. We discuss some significant categories as following.

For Android Applications. FlowDroid [5] is a widely used precise static taint analysis designed for Android applications, similarly, many works utilize static taint analysis to detect information leakage in Android apps [17, 21].

For Web Applications. Many works focus on discovering taint style vulnerabilities in the web applications [6, 14, 23, 24, 33, 35, 37], which is very different from the main target of SUTURE (*i.e.,* the kernel written in C). However, it is worth noting that Dahse *et al.* [14] proposes to detect the second-order vulnerability in the web applications, where the taint flows through some persistent data stores (*e.g.,* databases and files) on the server. Compared to it, SUTURE targets the more complex kernel and the general global states as taint relays, supporting taint flows of arbitrary order.

For Binaries. iDEA [7] utilizes taint tracking to find vulnerabilities in Apple kernel driver binaries. Cova *et al.* [13], DTaint [10] and Saluki [18] perform binary level static taint analysis based on symbolic execution to find vulnerabilities in the executable. SUTURE assumes the source code availability, which can benefit the accuracy (*e.g.,* the exact structure layout). Besides, symbolic execution may not scale well for the large code base like the kernel, especially when hunting the high-order vulnerabilities.

For Open-Source C Programs. This is the most relevant category to SUTURE due to the same target. Chen *et al.* [9] uses static taint analysis to discover the implicit information leak in the kernel network stack, Zhang *et al.* [41] and Unisan [25] try to discover uninitialized memory allocations, KINT [36] can detect the integer errors in kernel and user programs. Dr. Checker [26] proposes a static analysis framework to discover different taint-style vulnerabilities. Johnson *et al.* [22] and UBITECT [40] adopt type inference based methods to detect specific taint vulnerabilities. Compared to these works, SUTURE has multiple enhancements (§3) on the static taint analysis to make it more precise, and more importantly, supports the high-order taint analysis. Yamaguchi *et al.* [39] and Shastry *et al.* [29] try to automatically infer the taint vulnerability patterns from known instances, and use them to search similar vulnerabilities. SUTURE on the other hand discovers new taint vulnerabilities from the ground.

Improvements on Static Analysis. Many works focus on improving the static analysis precision and/or efficiency, we discuss some of them as following.

Taint Analysis. TAINTINDUCE [12] automatically infers the taint propagation rules from dynamic analysis to avoid the inaccuracies of the human defined rules, for the same sake, Neutaint [30] employs neural network to conduct the dynamic taint analysis. ConDySTA [43] uses dynamic analysis results to improve static taint analysis accuracy. P/Taint [19] tries to unify the taint and points-to analysis to ease the implementation.

Path-Sensitivity. ESP [15] improves the scalability of the path-sensitive analysis by merging branches leading to same program states of interest, for the same goal, Fusion [32] makes the SMT solver work directly on the program dependence graph, together with the static analysis. Dillig *et al.* [16] improve path-sensitivity by considering the variable observability and the necessary and sufficient conditions of original path constraints. SUTURE's opportunistic path-sensitivity is more lightweight and mainly designed for trimming infeasible paths efficiently.

Other Improvements. Pearce *et al.* [27] extends the set-constraints language to support an efficient field-sensitive pointer analysis for C, Saturn [38] builds its static bug detection on boolean satisfiability (SAT) for a better precision and scalability. Heo *et al.* [20] uses machine learning to guide the switch between sound and unsound static analysis, taking the best of both worlds. Pinpoint [31] defers the inter-procedure flow construction to the bug detection phase (on-demand) for a better efficiency, analogously, SUTURE also constructs the cross-entry flows in an demand-driven way (§4).

In general, we consider these works complementary to SUTURE, as they can potentially help improve the precision and efficiency of SUTURE's static analysis.

9 CONCLUSION

In this work, we develop SUTURE, a precise static analysis tool that can be used to discover complex high-order taint vulnerabilities in large code bases like the Linux kernel, a goal that was previously not attempted via static analysis. SUTURE successfully discovers new severe high-order vulnerabilities in the kernel, with a reasonable accuracy as perceived by the warning reviewers and an acceptable performance.

ACKNOWLEDGMENTS

We thank our shepherd Deian Stefan and anonymous reviewers for their helpful comments. We also thank Lingtong Shen for the useful discussion on high-order vulnerabilities. This work is partially supported by NSF grant #1652954.

REFERENCES

[1] 2021. CVE-2020-7053. https://nvd.nist.gov/vuln/detail/CVE-2020-7053.
[2] 2021. CVE-2020-8648. https://nvd.nist.gov/vuln/detail/CVE-2020-8648.
[3] 2021. Syzkaller. https://opensource.google/projects/syzkaller.
[4] 2021. The patch for a high-order taint vulnerability in Qualcomm driver. https://review.lineageos.org/c/LineageOS/android_kernel_motorola_msm8953/+/169169.
[5] 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. *Acm Sigplan Notices* 49, 6, 259–269.
[6] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 334–349.
[7] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. 2020. idea: Static analysis on the security of apple kernel drivers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1185–1202.
[8] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: a static/symbolic tool for finding good bugs in good (browser) code. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 199–216.
[9] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. 2015. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 388–400.
[10] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 430–441.
[11] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. 1996. Effective representation of aliases and indirect memory operations in SSA form. In *International Conference on Compiler Construction*. Springer, 253–267.
[12] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics.. In *NDSS*.
[13] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. 2006. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 269–278.
[14] Johannes Dahse and Thorsten Holz. 2014. Static detection of second-order vulnerabilities in web applications. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 989–1003.
[15] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. 57–68.
[16] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 270–280.
[17] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 576–587.
[18] Ivan Gotovchits, Rijnard Van Tonder, and David Brumley. 2018. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, Vol. 2018.
[19] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.
[20] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 519–529.
[21] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 106–117.
[22] Rob Johnson and David Wagner. 2004. Finding user/kernel pointer bugs with type inference.. In *USENIX Security Symposium*, Vol. 2. 0.
[23] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.
[24] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*. 27–36.
[25] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 920–932.
[26] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1007–1024.
[27] David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 4–es.
[28] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.
[29] Bhargava Shastry, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. 2017. Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing.. In *WOOT*.
[30] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1527–1543.
[31] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–706.
[32] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 930–943.
[33] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1053–1068.
[34] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.
[35] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.
[36] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Improving integer security for systems with {KINT}. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 163–177.
[37] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 171–180.
[38] Yichen Xie and Alex Aiken. 2007. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3 (2007), 16–es.
[39] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.
[40] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 221–232.
[41] Hang Zhang, Dongdong She, and Zhiyun Qian. 2016. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1663–1674.
[42] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1205–1220.
[43] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. 2021. ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 796–812.