DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers

Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna, *UC Santa Barbara*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers

Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens,
Christopher Kruegel, and Giovanni Vigna
{*machiry, cspensky, jcorina, stephens, chris, vigna*}*@cs.ucsb.edu*
University of California, Santa Barbara

Abstract

While kernel drivers have long been know to poses huge security risks, due to their privileged access and lower code quality, bug-finding tools for drivers are still greatly lacking both in quantity and effectiveness. This is because the pointer-heavy code in these drivers present some of the hardest challenges to static analysis, and their tight coupling with the hardware make dynamic analysis infeasible in most cases. In this work, we present DR. CHECKER, a *soundy* (i.e., mostly sound) bug-finding tool for Linux kernel drivers that is based on well-known program analysis techniques. We are able to overcome many of the inherent limitations of static analysis by scoping our analysis to only the most bug-prone parts of the kernel (i.e., the drivers), and by only sacrificing soundness in very few cases to ensure that our technique is both scalable and precise. DR. CHECKER is a fully-automated static analysis tool capable of performing general bug finding using both pointer and taint analyses that are flow-sensitive, context-sensitive, and field-sensitive on kernel drivers. To demonstrate the scalability and efficacy of DR. CHECKER, we analyzed the drivers of nine production Linux kernels (3.1 million LOC), where it correctly identified 158 critical zero-day bugs with an overall precision of 78%.

1 Introduction

Bugs in kernel-level code can be particularly problematic in practice, as they can lead to severe vulnerabilities, which can compromise the security of the entire computing system (e.g., Dirty COW [5]). This fact has not been overlooked by the security community, and a significant amount of effort has been placed on verifying the security of this critical code by means of manual inspection and both static and dynamic analysis techniques. While manual inspection has yielded the best results historically, it can be extremely time consuming, and is quickly becoming intractable as the complexity and volume of kernel-level code increase. Low-level code, such as kernel drivers, introduce a variety of hard problems that must be overcome by dynamic analysis tools (e.g., handling hardware peripherals). While some kernel-level dynamic analysis techniques have been proposed [23, 25, 29, 46], they are ill-suited for bug-finding as they were implemented as kernel monitors, not code verification tools. Thus, static source code analysis has long prevailed as the most promising technique for kernel code verification and bug-finding, since it only requires access to the source code, which is typically available.

Unfortunately, kernel code is a worst-case scenario for static analysis because of the liberal use of pointers (i.e., both function and arguments are frequently passed as pointers). As a result, tool builders must make the tradeoff between *precision* (i.e., reporting too many false positives) and *soundness* (i.e., reporting all true positives). In practice, precise static analysis techniques have struggled because they are either computationally infeasible (i.e., because of the state explosion problem), or too specific (i.e., they only identify a very specific type of bug). Similarly, sound static analysis techniques, while capable of reporting all bugs, suffer from extremely high false-positive rates. This has forced researchers to make variety of *assumptions* in order to implement practical analysis techniques. One empirical study [14] found that users would ignore a tool if its false positive rate was higher than 30%, and would similarly discredit the analysis if it did not yield valuable results early in its use (e.g., within the first three warnings).

Nevertheless, numerous successful tools have been developed (e.g., Coverity [14], Linux Driver Verification [36], APISan [64]), and have provided invaluable insights into both the types and locations of bugs that exist in critical kernel code. These tools range from precise, unsound, tools capable of detecting very specific classes of bugs (e.g., data leakages [32], proper `fprintf` usage [22], user pointer deferences [16]) to sound, im-

precise, techniques that detect large classes of bugs (e.g., finding all usages of `strcpy` [55]). One notable finding early on was that a disproportionate number of errors in the kernel were found in the drivers, or modules. It was shown that drivers accounted for seven times more bugs than core code in Linux [19] and 85% of the crashes in Windows XP [49]. These staggering numbers were attributed to lower overall code quality in drivers and improper implementations of the complex interactions with the kernel core by the third party supplying the driver.

In 2011, Palix *et al.* [39] analyzed the Linux kernel again and showed that while drivers still accounted for the greatest number of bugs, which is likely because drivers make up 57% of the total code, the fault rates for drivers where no longer the highest. Our recent analysis of main line linux kernel commit messages found that 28% of CVE patches to the linux repository in the past year involved kernel drivers (19% since 2005), which is in line with previous studies [17]. Meanwhile, the mobile domain has seen an explosion of new devices, and thus new drivers, introduced in recent years. The lack of attention being paid to these drivers, and their potential danger to the security of the devices, has also not gone unnoticed [47]. Recent studies even purport that mobile kernel drivers are, again, the source of up to 85% of the reported bugs in the Android [48] kernel. Yet, we are unaware of any large-scale analysis of these drivers.

In this work, we present DR. CHECKER, a fully-automated static-analysis tool capable of identifying numerous classes of bugs in Linux kernel drivers. DR. CHECKER is implemented as a completely modular framework, where both the types of analyses (e.g., points-to or taint) and the bug detectors (e.g., integer overflow or memory corruption detection) can be easily augmented. Our tool is based on well-known program analysis techniques and is capable of performing both pointer and taint analysis that is flow-, context-, and field-sensitive. DR. CHECKER employs a *soundy* [31] approach, which means that our technique is mostly sound, aside from a few well-defined assumptions that violate soundness in order to achieve a higher precision. DR. CHECKER, is the first (self-proclaimed) *soundy* static-analysis-based bug-finding tool, and, similarly, the first static analysis tool capable of large-scale analysis of general classes of bugs in driver code. We evaluated DR. CHECKER by analyzing nine popular mobile device kernels, 3.1 million lines of code (LOC), where it correctly reported 3,973 flaws and resulted the discovery of **158** [6–10] previously *unknown* bugs. We also compared DR. CHECKER against four other popular static analysis tools, where it significantly outperformed all of them both in detection rates and total bugs identified. Our results show that DR. CHECKER not only produces useful results, but does so with extremely high precision (78%).

In summary, we claim the following contributions:

- We present the first *soundy* static-analysis technique for pointer and taint analysis capable of large-scale analysis of Linux kernel drivers.
- We show that our technique is capable of flow-sensitive, context-sensitive, and field-sensitive analysis in a pluggable and general way that can easily be adapted to new classes of bugs.
- We evaluated our tool by analyzing the drivers of nine modern mobile devices, which resulted in the discovery of 158 *zero-day* bugs.
- We compare our tool to the existing state-of-the-art tools and show that we are capable of detecting more bugs with significantly higher precision, and with high-fidelity warnings.
- We are releasing DR. CHECKER as an open-source tool at `github.com/ucsb-seclab/dr_checker`.

2 Background

Kernel bug-finding tools have been continuously evolving as both the complexity and sheer volume of code in the world increases. While manual analysis and `grep` may have been sufficient for fortifying the early versions of the Linux kernel, these techniques are neither scalable nor rigorous enough to protect the kernels that are on our systems today. Ultimately, all of these tools are developed to raise *warnings*, which are then examined by a human analyst. Most of the initial, and more successful bug-finding tools were based on `grep`-like functionality and pattern matching [45, 55, 57]. These tools evolved to reduce user interaction (i.e., removing the need for manual annotation of source code) by using machine learning and complex data structures to automatically identify potential dangerous portions of code [41, 59–63]. While these tools have been shown to return useful results, identifying a number of critical bugs, most of them are developed based on empirical observation, without strong formal guarantees.

Model checkers (e.g., SLAM [13], BLAST [27], MOPS [18]) provide much more context and were able to provide more formalization, resulting in the detection of more interesting flaws. However, these techniques soon evolved into more rigorous tools, capable of more complex analyses (e.g., path-sensitive ESP [22]) and the more recent tools are capable of extracting far more information about the programs being analyzed to perform even more in-depth analysis (e.g., taint analysis [61]). While some have been implemented on top of custom tools and data structures (e.g., Joern [59–62]),

others have been implemented as compiler-level optimizations on top of popular open-source projects (e.g., LLVM [32]). In all cases, these tools are operating on abstract representations of the program, such as the abstract syntax tree (AST) or the control flow graph (CFG), which permit a more rigorous formal analysis of the properties of the program.

Motivation. Before delving into the details of DR. CHECKER, we first present a motivating example in the form of a bug that was discovered by DR. CHECKER. In this bug, which is presented in Listing 1, a tainted structure is copied in from userspace using `copy_from_user`. A size field of this structure is then multiplied by the size of another driver structure (`flow_p.cnt * sizeof(struct bst_traffic_flow_prop)`), which is vulnerable to an *integer overflow*. This bug results in a much smaller buffer being allocated that would actually be required for the data. This overflow would not be particularly problematic if it wasn't for the fact that the originally tainted length (i.e., the very large number) is later used to determine how much data will be copied in the buffer (`adjust_traffic_flow_by_pkg(buf, flow_p.cnt)`), resulting in memory corruption.

There are many notable quirks in this bug that make it prohibitively difficult for naïve static analysis techniques. First, the bug arises from tainted-data (i.e., `argp`) propagating through multiple usages into a dangerous function, which is only detectable by a flow-sensitive analysis. Second, the integer overflow occurs because of a specific field in the user-provided `struct`, not the entire buffer. Thus, any analysis that is not field sensitive would over-approximate this and incorrectly identify `flow_p` as the culprit. Finally, the memory corruption in a different function (i.e., `adjust_traffic_flow_by_pkg`), which means that that the analysis must be able to handle inter-procedural calls in a context-sensitive way to precisely report the origin of the tainted data. Thus, this bug is likely only possible to detect and report concisely with an analysis that is flow-, context-, and field-sensitive. Moreover, the fact that this bug exists in the driver of a popular mobile device, shows that it evaded both expert analysts and possibly existing bug-finding tools.

Listing 1: An integer overflow in Huawei's Bastet driver that was discovered by DR. CHECKER

```
struct bst_traffic_flow_pkg {
    uint32_t cnt;
    uint8_t value[0];
};
...
uint8_t *buf = NULL;
int buf_len = 0;
struct bst_traffic_flow_pkg flow_p;

if (copy_from_user(&flow_p, argp,
    sizeof(struct bst_traffic_flow_pkg))) {
  break;
}

if (0 == flow_p.cnt) {
  bastet_wake_up_traffic_flow();
  rc = 0;
  break;
}

// ** Integer overflow bug **
// e.g., 0x80000001 * 0x20 = 0x20
buf_len = flow_p.cnt *
          sizeof(struct bst_traffic_flow_prop);
buf = (uint8_t *)kmalloc(buf_len, GFP_KERNEL);
if (NULL == buf) {
  BASTET_LOGE("kmalloc failed");
  rc = -ENOMEM;
  break;
}

if (copy_from_user(buf,
    argp + sizeof(struct bst_traffic_flow_pkg),
    buf_len)) {
  BASTET_LOGE("pkg copy_from_user error");
  kfree(buf);
  break;
}
//  Modifies flow_p.cnt, not buf_len, bytes in buf!
rc = adjust_traffic_flow_by_pkg(buf, flow_p.cnt);
...
```

3 Analysis Design

DR. CHECKER uses a modular interface for its analyses. This is done by performing a general analysis pass over the code, and invoking *analysis clients* at specific points throughout the analysis. These analysis clients all share the same global state, and benefit from each other's results. Once the analysis clients have run and updated the global state of the analysis, we then employ numerous *vulnerability detectors*, which identify specific properties of known bugs and raise warnings (e.g., a tainted pointer was used as input to a dangerous function). The general architecture of DR. CHECKER is depicted in Figure 1, and the details of our analysis and vulnerability detectors are outlined in the following sections.

Below we briefly outline a few of our core assumptions that contribute to our *soundy* analysis design:

Assumption 1. We assume that all of the code in the mainline Linux core is implemented *perfectly*, and we do not perform any inter-procedural analysis on any kernel application program interface (API) calls.

Assumption 2. We only perform the number of traversals required for a reach-def analysis in loops, which could result in our points-to analysis being unsound.

Assumption 3. Each call instruction will be traversed only once, even in the case of loops. This is to avoid creating additional contexts and limit false positives, which may result in our analysis being unsound.

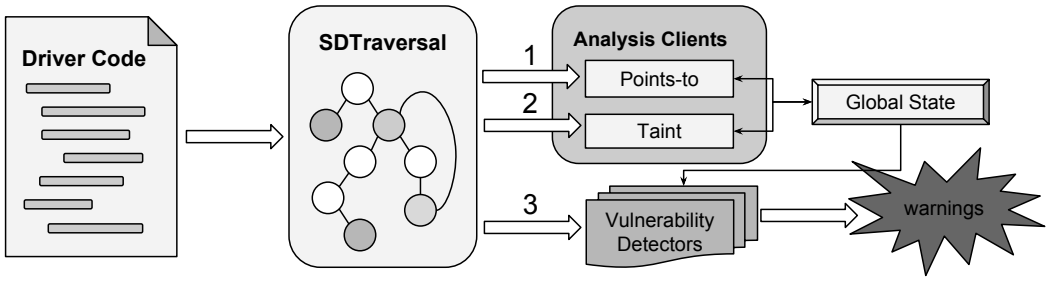

Figure 1: Pluggable static analysis architecture implemented by DR. CHECKER.

3.1 Terminology and Definitions

In this section we define the various terms and concepts that we use in the description of our analysis.

Definition 3.1. A *control flow graph (CFG)* of a function is a directed graph where each node represents a *basic block* (i.e., a contiguous sequence of non-branch instructions) and the edges of the graph represent possible control flow between the basic blocks.

Definition 3.2. A *strongly connected component (SCC)* of a graph is a sub-graph, where there exists a bi-directional path between any pair of nodes (e.g., a loop).

Definition 3.3. *Topological sort or ordering* of nodes in a directed graph is an ordering of nodes such that, for every edge from node v to u, v is traversed before u. While this is well-defined for acyclic graphs, it is less straightforward for cyclic graphs (e.g., a CFG with loops). Thus, when performing a topological sort on a CFG, we employ Tarjan's algorithm [50], which instead topologically sorts the SCCs.

Definition 3.4. An *entry* function, ε, is a function that is called with at least one of its arguments containing tainted data (e.g., an `ioctl` call).

Definition 3.5. The *context*, Δ, of a function in our analysis is an ordered list of call sites (e.g., function calls on the stack) starting from an entry function. This list indicates the sequence of function calls and their locations in the code that are required to reach the given function. More precisely, $\Delta = \{\varepsilon, c_1, c_2, ...\}$ where c_1 is call made from within the entry function (ε) and for all $i > 1$, c_i is a call instruction in the function associated with the previous call instruction (c_{i-1}).

Definition 3.6. The *global taint trace map*, τ, contains the information about our tainted values in the analysis. It maps a specific value to the sequence of instructions (I) whose execution resulted in the value becoming tainted.

$$\tau : \begin{cases} v \rightarrow \{I_1, I_2, I_3, ...\} & \text{if TAINTED} \\ v \rightarrow \emptyset & \text{otherwise} \end{cases}$$

Definition 3.7. An *alias object*, $\hat{a} = \{\rho, t\}$, is a tuple that consists of a map (ρ) between offsets into that object, n, and the other corresponding alias objects that those offsets can point to, as well as a local taint map (t) for each offset. For example, this can be used to represent a structure stored in a static location, representing an alias object, which contains pointers at given offsets (i.e., offsets into that object) to other locations on the stack (i.e., their alias objects). More precisely, $\rho : n \rightarrow \{\hat{a_1}, \hat{a_2}, \hat{a_3}, ...\}$ and $t : n \rightarrow \{I_1, I_2, I_3, ...\}$. We use both $\hat{a}(n)$ and $\rho(n)$ interchangeably, to indicate that we are fetching all of the alias objects that could be pointed to by a field at offset n. We use $\hat{a}_t$ to refer to the taint map of location $\hat{a}$, and similarly $\hat{a}_t(n)$ to refer to taint at a specific offset. These maps allow us to differentiate between different fields of a structure to provide field-sensitivity in our analysis.

The following types of locations are traced by our analysis:

1. Function local variables (or stack locations): We maintain an alias object for each local variable.
2. Dynamically allocated variables (or heap locations): These are the locations that are dynamically allocated on the program heap (e.g., as retrieved by `malloc` or `get_page`). We similarly create one alias object for each allocation site.
3. Global variables: Each global variable is assigned a unique alias object.

Stack and heap locations are both context-sensitive (i.e., multiple invocations of a function with different contexts will have different alias objects). Furthermore, because of our context propagation, heap locations are call-site sensitive (i.e., for a given context, one object will be created for each call site of an allocation function).

Definition 3.8. Our *points-to* map, ϕ, is the map between a value and all of the possible locations that it can point to, represented as a set of tuples containing alias objects and offsets into those objects.

$$\phi : v \rightarrow \{(n_1, \hat{a_1}), (n_1, \hat{a_2}), (n_2, \hat{a_3}), ...\}$$

For example, consider the instruction `val1 = &info->dirmap`, where `info` represents a structure on the stack and member `dirmap` is at offset 8. This instruction would result in the value (`val1`) pointing to the offset 8 within the alias object `info` (i.e., $\phi(\texttt{val1}) = \{(8, \texttt{info})\}$).

Definition 3.9. The Global State, S, of our analysis contains all of the information computed for every function, at every context. We define it as

$$S = \{\phi_c, \tau_c\},$$

where $\phi_c : \Delta \rightarrow \phi$ is the map between a context and the corresponding points-to map, and $\tau_c : \Delta \rightarrow \tau$ is the map between a context and corresponding taint trace map.

3.2 Soundy Driver Traversal (SDT)

While most of the existing static analysis techniques [13, 28] run their abstract analysis until it reaches a fixed-point before performing bug detection, this can be problematic when running multiple analyses, as the different analyses may not have the same precision. Thus, by performing analysis on the post-completion results, these tools are fundamentally limiting the precision of all of their analyses to the precision of the *least* precise analysis. To avoid this, and ensure the highest precision for all of our analysis modules, we perform a *flow-sensitive* and *context-sensitive* traversal of the driver starting from an entry point. Our specific analysis modules (i.e., taint and points-to) are implemented as *clients* in this framework, and are invoked with the corresponding context and current global state as the code is being traversed. This also allows all of the analyses, or clients, to consume each other's results whenever the results are needed, and without loss of precision. Moreover, this allows us to perform a single traversal of the program for all of the underlying clients.

It is important to note that some of the client analyses may actually need more traversals through the CFG than others to reach a fixed point. For example, a points-to analysis might need more traversals through a loop to reach a fixed point than a taint analysis. However, our code exploration is analysis-agnostic, which means we must ensure that we always perform the maximum number of traversals required by all of our analyses. To ensure this property, we use reach-def analysis [38] as a baseline (i.e., we traverse the basic blocks such that a reaching definition analysis will reach a fixed point). This ensures that all of the writes that can reach an instruction directly will be reached. This means that our points-to analysis may not converge, as it would likely require far more iterations. However, in the worst case, points-to analysis could potentially grow unconstrained, resulting in everything pointing to everything. Thus, we make this necessary sacrifice to soundness to ensure convergence and a practical implementation.

Algorithm 1: Soundy driver traversal analysis

```
function SDTraversal((S, Δ, F))
    sccs ← topo_sort(CFG(F))
    forall the scc ∈ sccs do
        if is_loop(scc) then
            HandleLoop(S, Δ, scc)
        else
            VisitSCC(S, Δ, scc)
        end
    end

function VisitSCC((S, Δ, scc))
    forall the bb ∈ scc do
        forall the I ∈ bb do
            if is_call(I) then
                HandleCall(S, Δ, I)
            else
                if is_ret(I) then
                    S ← S ∪ {φ_Δ(ret_val), τ_Δ(ret_val)}
                else
                    DispatchClients(S, Δ, I)
                end
            end
        end
    end

function HandleLoop((S, Δ, scc))
    num_runs ← LongestUseDefChain(scc)
    while num_runs ≠ 0 do
        VisitSCC(S, Δ, scc)
        num_runs ← num_runs − 1
    end

function HandleCall((S, Δ, I))
    if ¬is_visited(S, Δ, I) then
        targets ← resolve_call(I)
        forall the f ∈ targets do
            Δ_new ← Δ||I
            φ_new ← (Δ_new → (φ_c(Δ)(args), φ_c(Δ)(globals)))
            τ_new ← (Δ_new → (τ_c(Δ)(args), τ_c(Δ)(globals)))
            S_new ← {φ_new, τ_new}
            SDTraversal(S_new, Δ_new, f)
        end
        mark_visited(S, Δ, I)
    end
```

Loops. When handling loops, we must ensure that we iterate over the loop enough times to ensure that every possible assignment of every variable has been exercised. Thus, we must compute the number of iterations needed

for a reach-def analysis to reach a fix-point on the loop and then perform the corresponding number of iterations on all the basic blocks in the loop. Note that, the number of iterations to converge on a loop for a standard reach-def analysis is upper-bounded by the longest use-def chain in the loop (i.e., the longest number of instructions between the assignment and usage of a variable). The intuition behind this is that, in the worst case, every instruction could potentially depend on the variable in the use-def chain, such that their potential values could update in each loop. However, this can only happen as many times as their are instructions, since an assignment can only happen once per instruction.

Function calls. If a function call is a direct invocation and the target function is within the code that we are analyzing (i.e., it is part of the driver), it will be traversed with a new context (Δ_{new}), and the state will be both updated with a new points-to map (ρ_{new}) and a new taint trace map (τ_{new}), which contains information about both the function arguments and the global variables. For indirect function calls (i.e., functions that are invoked via a pointer), we use type-based target resolution. That is, given a function pointer of type $a =$ `(rettype)(arg1Type, arg2Type,..)`, we find all of the matching functions in the same driver that are referenced in a non-call instruction (e.g., `void *ptr = &fn`). This is implemented as the function *resolve_call* in Algorithm 1. Each call site or call instruction will be analyzed only once per context. We do not employ any special handlers for recursive functions, as recursion is rarely used in kernel drivers.

The complete algorithm, `SDTraversal`, is depicted in Algorithm 1. We start by topologically sorting the CFG of the function to get an ordered list of SCCs. Then, each SCC is handled differently, depending on whether it is a loop or not. Every SCC is traversed at the basic-block level, where every instruction in the basic block is provided to all of the possible clients (i.e., taint and points-to), along with the context and global state. The client analyses can collect and maintain any required information in the global state, making the information immediately available to each other.

To analyze a driver entry point ε, we first create an initial state: $S_{start} = \{\phi_{start}, \emptyset\}$, where ϕ_{start} contains the points-to map for all of the global variables. We then traverse all of the `.init` functions of the driver (i.e., the functions responsible for driver initialization [44]), which is where drivers will initialize most of their global objects. The resulting initialized state (S_{init}) is then appended with the taint map for any tainted arguments ($S_{init} = S_{init} \cup \tau_{init}$). We describe how we determine these tainted arguments in Section 5.3. Finally, we invoke our traversal on this function, `SDTraversal(`S_{init}`,` Δ_{init}`,` ε`)`, where the context $\Delta_{init} = \{e\}$.

We use the low-level virtual machine (LLVM) intermediate representation (IR), Bitcode [30], as our IR for analysis. Bitcode is a typed, static single assignment (SSA) IR, and well-suited for low-level languages like C. The analysis clients interact with our soundy driver traversal (SDT) analysis by implementing visitors, or transfer functions, for specific LLVM IR instructions, which enables them to both use and update the information in the global state of the analysis. The instructions that we define transfer functions for in the IR are:

1. *Alloca* (`v = alloca typename`) allocates a stack variable with the size of the type `typename` and assigns the location to v (e.g., `%1 = alloca i32`). SDT uses the instruction location to reference the newly allocated instruction. Since SDT is context-sensitive, the instruction location is a combination of the current context and the instruction offset within the function bitcode.

2. *BinOp* (`v = op op1, op2`) applies `op` to `op1` and `op2` and assigns the result to `v` (e.g., `%1 = add val, 4`). We also consider, the flow-merging instruction in SSA, usually called `phi` [21], to be the same as a binary operation. Since SDT is not path-sensitive, this does not affect the soundness.

3. *Load* (`v = load typename op`) is the standard load instruction, which loads the contents of type `typename` from the address represented by the operand `op` into the variable `v` (e.g., `%tmp1 = load i32* %tmp`).

4. *Store* (`store typename v, op`) is the standard store instruction, which stores the contents of type `typename` represented by the value `v` into the address represented by `op` (e.g., `store i8 %frombool1, %y.addr`).

5. *GetElementPtr (GEP)* is the instruction used by the IR to represent structure and array-based accesses and has fairly complex semantics [53]. A simplified way to represent this is `v = getelementptr typename ob, off`, which will get the address of the field at index `off` from the object `ob` of type `typename`, and store the referenced value in `v` (e.g., `%val = getelementptr %struct.point %my_point, 0`).

Both our points-to and taint analysis implement transfer functions based on these five instructions.

Algorithm 2: Points-to analysis transfer functions

function `updatePtoAlloca` $(\phi_c, \tau_c, \delta, I, v, loc_x)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $loc_x \leftarrow (x, \emptyset, \emptyset)$
  $map_{pt}(v) \leftarrow (0, loc_x)$
function `updatePtoBinOp` $(\phi_c, \tau_c, \delta, I, v, op_1, op_2)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_1 \leftarrow map_{pt}(op_1)$
  $pto_2 \leftarrow map_{pt}(op_2)$
  $set_1 \leftarrow \{(0, ob) \mid \forall(\_, ob) \in pto_1\}$
  $set_2 \leftarrow \{(0, ob) \mid \forall(\_, ob) \in pto_2\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_1 \cup set_2$
function `updatePtoLoad` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_1 \leftarrow \{ob(n) \mid \forall(n, ob) \in pto_{op}\}$
  $set_2 \leftarrow \{(0, ob) \mid \forall ob \in set_1\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_2$
function `updatePtoStore` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $pto_v \leftarrow map_{pt}(v)$
  $set_v \leftarrow \{ob \mid \forall(\_, ob) \in pto_v\}$
  $\forall(n, ob) \in pto_{op}$ **do** $ob(n) \leftarrow ob(n) \cup set_v$
function `updatePtoGEP` $(\phi_c, \tau_c, \delta, I, v, op, off)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_{op} \leftarrow \{ob(n) \mid \forall(n, ob) \in pto_{op}\}$
  $set_v \leftarrow \{(off, ob) \mid \forall ob \in set_{op}\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_v$

Algorithm 3: Taint analysis transfer functions

function `updateTaintAlloca` $(\phi_c, \tau_c, \delta, I, v, loc_x)$
  Nothing to do

function `updateTaintBinOp` $(\phi_c, \tau_c, \delta, I, v, op_1, op_2)$
  $map_t \leftarrow \tau_c(\delta)$
  $set_v \leftarrow map_t(op_1) \cup map_t(op_2)$
  $map_t(v) \leftarrow set_v || I$
function `updateTaintLoad` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_{op} \leftarrow \{ob_t(n) || I \mid \forall(n, ob) \in pto_{op}\}$
  $map_t \leftarrow \tau_c(\delta)$
  $map_t(v) \leftarrow map_t(v) \cup set_{op}$
function `updateTaintStore` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $map_t \leftarrow \tau_c(\delta)$
  $tr_v \leftarrow map_t(v)$
  $\forall(n, ob) \in pto_{op}$ **do** $ob_t(n) \leftarrow ob_t(n) \cup (tr_v || I)$
function `updateTaintGEP` $(\phi_c, \tau_c, \delta, I, v, op, off)$
  UPDATETAINTBINOP$(\phi_c, \tau_c, \delta, I, v, op, off)$

3.3 Points-to Analysis

The result of our points-to analysis is a list of values and the set of all of the possible objects, and offsets, that they can point to. Because of the way in which we constructed our *alias location objects* and transfer functions, we are able to ensure that our points-to results are field-sensitive. That is, we can distinguish between objects that are pointed to by different fields of the same object (e.g., different elements in a `struct`). Thus, as implemented in SDT, we are able to obtain points-to results that are flow-, context-, and field-sensitive.

Dynamic allocation. To handle dynamic allocation in our points-to analysis, we maintain a list of kernel functions that are used to allocate memory on the heap (e.g., `__kmalloc`, `kmem_cache_alloc`, `get_free_page`). For each call-site to these functions, we create a unique *alias object*. Thus, for a given context of a function, each allocation instruction has a single `alias location`, regardless of the number of times that it is visited. For example, if there is a call to `kmalloc` in the basic block of a loop, we will only create one alias location for it.

Internal kernel functions. Except for few kernel API functions, whose effects can be easily handled (e.g., `memcpy`, `strcpy`, `memset`), we ignore all of the other kernel APIs and core kernel functions. For example, if the target of a call instruction is the function `i2c_master_send`, which is part of the core kernel, we do not follow the call. Contrary to the other works, which check for valid usage of kernel API functions [12,64], we assume that all usages of these functions are valid, as we are only concerned with analyzing the more error-prone driver code. Thus, we do no follow any function calls into the core kernel code. While, we may miss some points-to information because of this, again sacrificing soundness, this assumption allows us to be more precise within the driver and scale our analysis.

The *update points-to* transfer functions (`updatePto*`) for the various instructions are as shown in Algorithm 2.

3.4 Taint Analysis

Taint analysis is a critical component of our system, as almost all of our bug detectors use its results. Similar to our points-to analysis, the results of our taint analysis are flow-, context-, and field-sensitive.

The *taint sources* in our analysis are the arguments of the entry functions. Section 5.3 explains the different types of entry functions and their correspondingly tainted arguments. We also consider special kernel functions that copy data from user space (e.g., `copy_from_user`, `simple_write_to_buffer`) as taint sources and taint all of the fields in the alias locations of the points-to map for

Listing 2: A buffer overflow bug detected in Mediatek's Accdet driver by ITDUD where `buf` is assumed to be a single character but the use of "%s" will continue reading the buffer until a null-byte is found.

```
static char call_status;
...
static ssize_t
  accdet_store_call_state
  (struct device_driver *ddri,
  const char *buf, size_t count)
{
  // ** Improper use of tainted data **
  // buf can contain more than one char!
  int ret = sscanf(buf, "%s", &call_status);

  // The return value is checked, but it's too late
  if (ret != 1) {
    ACCDET_DEBUG("accdet: Invalid values\n");
    return -EINVAL;
  }

  switch (call_status) {
  case CALL_IDLE:
  ...
}
```

the destination operands of these functions. Our *taint propagators* are implemented as various transformation functions (`updateTaint*` in Algorithm 3). Similar to our points-to analysis, we do not propagate taint for any core kernel function calls, aside from a few exceptions (e.g., `memcpy`). The *taint sinks* in our analysis are dependent on the vulnerability detectors, as every detector has its own taint policy. These detectors will raise warnings if any tainted data violates a specified policy (e.g., if a tainted value is used as the length in a `memcpy`).

4 Vulnerability Detectors

This section describes the various vulnerability detectors that were used in our analysis. These detectors are highly configurable and are able to act on the results from both our points-to and taint analysis. They are implemented as plugins that are run continuously as the code is being analyzed, and operate on the results from our analysis clients (i.e., taint and points-to analysis). Our architecture enables us to very quickly implement new analyses to explore new classes of vulnerabilities. In fact, in the process of analyzing our results for this paper, we were able to create the Global Variable Race Detector (GVRD) detector and deploy it in less than 30 minutes.

Almost all of the detectors use taint analysis results to verify a vulnerable condition and produce a taint trace with all of their emitted warnings. The warnings also provide the line numbers associated with the trace for ease of triaging. The various bug detectors used by DR. CHECKER in our analysis are explained below:

Improper Tainted-Data Use Detector (ITDUD) checks for tainted data that is used in risky functions (i.e., `strc*`, `strt*`, `sscanf`, `kstrto`, and `simple_strto` family functions). An example of a previously unknown buffer overflow, detected via ITDUD, is shown in Listing 2.

Listing 3: A zero-day vulnerability discovered by DR. CHECKER in Mediatek's mlog driver using our TAD and TLBD analysis. First TAD identified an integer overflow bug (`len - MLOG_STR_LEN`). TLBD then identified that this tainted length was being used as a bound condition for the while loop where data is being copied into kernel space.

```
#define MLOG_STR_LEN          16
...
int mlog_doread(char __user *buf, size_t len)
{
  unsigned i;
  int error = -EINVAL;
  char mlog_str[MLOG_STR_LEN];
  ...
  // len is unsigned
  if (!buf || len < 0)
    goto out;
  error = 0;
  // len not checked against MLOG_STR_LEN
  if (!len)
    goto out;
  // buf of len confirmed to be in user space
  if (!access_ok(VERIFY_WRITE, buf, len)) {
    error = -EFAULT;
    goto out;
  }
  ...
  i = 0;
  ...
  // ** Integer underflow bug **
  // len - MLOG_STR_LEN (16) can be negative
  // and is compared with unsigned i
  while (!error && (mlog_start != mlog_end)
         && i < len - MLOG_STR_LEN) {
    int size;
    ...
    size = snprintf(mlog_str, MLOG_STR_LEN,
            strfmt_list[strfmt_idx++], v);
    ...
    // this function is an unsafe copy
    // this results in writing past buf
    // potentially into kernel address space
    if (__copy_to_user(buf, mlog_str, size))
      error = -EFAULT;
    else {
      buf += size;
      i += size;
    }
  }
}
```

Tainted Arithmetic Detector (TAD) checks for tainted data that is used in operations that could cause an overflow or underflow (e.g., `add`, `sub`, or `mul`). An example of a zero-day detected by TAD is shown in Listing 3.

Invalid Cast Detector (ICD) keeps tracks of allocation sizes of objects and checks for any casts into an object of a different size.

Tainted Loop Bound Detector (TLBD) checks for tainted data that is used as a loop bound (i.e., a loop guard in which at least one of the values is tainted). These bugs could lead to a denial of service or even an arbitrary memory write. The example in Listing 3 shows this in a real-world bug, which also triggered on TAD.

Listing 4: An information leak bug via padded fields detected by our ULD in Mediatek's FM driver where a struct's memory is not sanitized before being copied back to user space leaking kernel stack data.

```
fm_s32 fm_get_aud_info(fm_audio_info_t *data)
{

  if (fm_low_ops.bi.get_aud_info) {
    return fm_low_ops.bi.get_aud_info(data);
  } else {
    data->aud_path = FM_AUD_ERR;
    data->i2s_info.mode = FM_I2S_MODE_ERR;
    data->i2s_info.status = FM_I2S_STATE_ERR;
    data->i2s_info.rate = FM_I2S_SR_ERR;
    return 0;
  }
}
...
  case FM_IOCTL_GET_AUDIO_INFO:
    fm_audio_info_t aud_data;
    // ** no memset of aud_data **
    // Not all fields of aud_data are initialized
    ret = fm_get_aud_info(&aud_data);
    if (ret) {
      WCN_DBG(FM_ERR|MAIN, "fm_get_aud_info err\n");
    }
    // Copying the struct results in data-leakage
    // from padding and uninitialized fields
    if (copy_to_user((void *)arg, &aud_data,
        sizeof(fm_audio_info_t))) {
      WCN_DBG(FM_ERR|MAIN, "copy_to_user error\n");
      ret = -EFAULT;
      goto out;
    }
    ...
```

Tainted Pointer Dereference Detector (TPDD) detects pointers that are tainted and directly dereferenced. This bug arises when a user-specified index into a kernel structure is used without checking.

Tainted Size Detector (TSD) checks for tainted data that is used as a size argument in any of the `copy_to_` or `copy_from_` functions. These types of bugs can result in information leaks or buffer overflows since the tainted size is used to control the number of copied bytes.

Uninit Leak Detector (ULD) keeps tracks of which objects are initialized, and will raise a warning if any `src` pointer for a userspace copy function (e.g., `copy_to_user`) can point to any uninitialized objects. It also detects structures with padding [40] and will raise a warning if `memset` or `kzalloc` has not been called on the corresponding objects, as this can lead to an information leak. An example of a previously unknown bug detected by this detector is as shown in Listing 4

Global Variable Race Detector (GVRD) checks for global variables that are accessed without a mutex. Since the kernel is reentrant, accessing globals without synchronization can result in race conditions that could lead to time of check to time of use (TOCTOU) bugs.

5 Implementation

DR. CHECKER is built on top of LLVM 3.8 [30]. LLVM was chosen because of its flexibility in writing analyses, applicability to different architectures, and excellent community support. We used integer range analysis as implemented by Rodrigues *et al.* [42]. This analysis is used by our vulnerability detectors to verify certain properties (e.g., checking for an invalid cast).

We implemented DR. CHECKER as an LLVM module pass, which consumes: a bitcode file, an `entry function name`, and an `entry function type`. It then runs our SDT analysis, employing the various analysis engines and vulnerability detectors. Depending on the `entry function type`, certain arguments to the entry functions are tainted before invoking the SDT (See Section 5.3).

Because our analysis operates on LLVM bitcode, we must first identify and build all of the driver's bitcode files for a given kernel (Section 5.1). Similarly, we must identify all of the entry points in these drivers (Section 5.2) in order to pass them to our SDT analysis.

5.1 Identifying Vendor Drivers

To analyze the drivers independently, we must first differentiate driver source code files from that of the core kernel code. Unfortunately, there is no standard location in the various kernel source trees for driver code. Making the problem even harder, a number of the driver source files omit vendor copyright information, and some vendors even modify the existing sources directly to implement their own functionality. Thus, we employ a combination of techniques to identify the locations of the vendor drivers in the source tree. First, we perform a `diff` against the mainline sources, and compare those files with a referenced vendor's configuration options to search for file names containing the vendor's name. Luckily, each vendor has a code-name that is used in all of their options and most of their files (e.g., Qualcomm configuration options contain the string `MSM`, Mediatek is `MTK`, and Huawei is either `HISI` or `HUAWEI`), which helps us identify the various vendor options and file names. We do this for all of the vendors, and save the locations of the drivers relative to the source tree.

Once the driver files are identified, we compile them using clang [51] into both Advanced RISC Machine (ARM) 32 bit and 64 bit bitcode files. This necessitated a few non-trivial modifications to clang, as there are numerous GNU C Compiler (GCC) compiler options used by the Linux kernel that are not supported by clang (e.g., the `-fno-var-tracking-assignments` and `-Wno-unused-but-set-variable` options used by various Android vendors). We also added additional

compiler options to clang (e.g., `-target`) to aid our analysis. In fact, building the Linux kernel using LLVM is an ongoing project [52], suggesting that considerable effort is still needed.

Finally, for each driver, we link all of the dependent vendor files into a single bitcode file using `llvm-link`, resulting in a self-contained bitcode file for each driver.

5.2 Driver Entry Points

Linux kernel drivers have various ways to interact with the userspace programs, categorized by 3 operations: file [20], attribute [35], and socket [37].

File operations are the most common way of interacting with userspace. In this case, the driver exposes a file under a known directory (e.g., `/dev`, `/sys`, or `/proc`) that is used for communication. During initialization, the driver specifies the functions to be invoked for various operations by populating function pointers in a structure, which will be used to handle specific operations (e.g., `read`, `write`, or `ioctl`). The structure used for initialization can be different for each driver type. In fact, there are at least *86 different types of structures in Android kernels* (e.g., `struct snd_pcm_ops`, `struct file_operations`, or `struct watchdog_ops` [3]). Even worse, the entry functions can be at different offset in each of these structures. For example, the `ioctl` function pointer is at field 2 in `struct snd_pcm_ops`, and at field 8 in `struct file_operations`. Even for the same structure, different kernels may implement the fields differently, which results in the location of the entry function being different for each kernel. For example, `struct file_operations` on Mediatek's mt8163 kernel has its `ioctl` function at field 11, whereas on Huawei, it appears at field 9 in the structure.

To handle these eccentricities in an automated way, we used `c2xml` [11] to parse the header files of each kernel and find the offsets for possible entry function fields (e.g., `read` or `write`) in these structures. Later, given a bitcode file for a driver, we locate the different file operation structures being initialized, and identify the functions used to initialize the different entry functions.

Listing 5: An initialization of a file operations structure in the mlog driver of Mediatek

```
1 static const struct file_operations
2 proc_mlog_operations = {
3   .owner = NULL,
4   .llseek = NULL,
5   .read = mlog_read,
6   .poll = mlog_poll,
7   .open = mlog_open,
8   .release = mlog_release,
9   .llseek = generic_file_llseek,
10 };
```

Table 1: Tainted arguments for each driver entry function type wether they are directly and indirectly tainted.

Entry Type	Argument(s)	Taint Type
Read (*File*)	`char *`*buf*, `size_t` *len*	Direct
Write (*File*)	`char *`*buf*, `size_t` *len*	Direct
Ioctl (*File*)	`long` *arg*	Direct
DevStore (*Attribute*)	`const char *`*buf*	Indirect
NetDevIoctl (*Socket*)	`struct *`*ifreq*	Indirect
V4Ioctl	`struct v4l2_format *`*f*	Indirect

These serve as our entry points for the corresponding operations. For example, given the initialization as shown in Listing 5, and the knowledge that read entry function is at offset 2 (zero indexed), we mark the function `mlog_read` as a read entry function.

Attribute operations are operations usually exposed by a driver to read or write certain attributes of that driver. The maximum size of data read or written is limited to a single page in memory.

Sockets operations are exposed by drivers as a socket file, typically a UNIX socket, which is used to communicate with userspace via various socket operations (e.g., `send`, `recv`, or `ioctl`).

There are also other drivers in which the kernel implements a main wrapper function, which performs initial verification of the user parameters and *partially* sanitizes them before calling the corresponding driver function(s). An example of this can be seen in the `V4L2 Framework` [66], which is used for video drivers. For our implementation we consider only `struct v4l2_ioctl_ops`, which can be invoked by userspace via the wrapper function `video_ioctl2`.

5.3 Tainting Entry Point Arguments

An entry point argument can contain either *directly* tainted data (i.e., the argument is passed directly by userspace and never checked) or *indirectly* tainted data (i.e., the argument points to a kernel location, which contains the tainted data). All of the tainted entry point functions can be categorized in six categories, which are shown in Table 1, along with the type of taint data that their arguments represent.

An explicit example of directly tainted data is shown in Listing 6. In this snippet, `tc_client_ioctl` is an `ioctl` entry function, so argument 2 (`arg`) is directly tainted. Thus, the statement `char c=(char*)arg` would be dereferencing tainted data and is flagged as a warning. Alternatively, argument 2 (`ctrl`) in `iris_s_ext_ctrls` is a `V4Ioctl` and is indirectly tainted. As such, the dereference (`data = (ctrl->controls[0]).string`) is safe, but it would taint `data`.

Listing 6: Example of tainting different arguments where `tc_client_ioctl` has a directly tainted argument and `iris_s_ext_ctrls`'s argument is indirectly tainted.

```
static long tc_client_ioctl(struct file *file,
        unsigned cmd, unsigned long arg) {
        ...
        char c=(char*)arg
        ...
}
static int iris_s_ext_ctrls(struct file *file,
  void *priv, struct v4l2_ext_controls *ctrl) {
      ...
  char *data = (ctrl->controls[0]).string;
      ...
  char curr_ch = data[0];
}
```

6 Limitations

Because of the DR. CHECKER's soundy nature, it cannot find all the vulnerabilities in all drivers. Specifically, it will miss following types of vulnerabilities:

- *State dependent bugs:* Since DR. CHECKER is a stateless system, it treats each entry point independently (i.e., taint does not propagate between multiple entry points). As a result, we will miss any bugs that occur because of the interaction between multiple entry points (e.g., CVE-2016-2068 [4]).
- *Improper API usage:* DR. CHECKER assumes that all the kernel API functions are *safe and correctly used* (Assumption 1 in Section 3). Bugs that occur because of improper kernel API usage will be missed by DR. CHECKER. However, other tools (e.g., APISan [64]) have been developed for finding these specific types of bugs and could be used to complement DR. CHECKER.
- *Non-input-validation bugs:* DR. CHECKER specifically targets input validation vulnerabilities. As such, non-input validation vulnerabilities (e..g, side channels or access control bugs) cannot be detected.

7 Evaluation

To evaluate the efficacy of DR. CHECKER, we performed a large-scale analysis of the following nine popular mobile device kernels and their associated drivers (437 in total). The kernel drivers in these devices range from very small components (31 LOC), to much more complex pieces of code (240,000 LOC), with an average of 7,000 LOC per driver. In total, these drivers contained over 3.1 million lines of code. However, many of these kernels re-use the same code, which could result in analyzing the same entry point twice, and inflate our results. Thus, we have grouped the various kernels based on their underlying chipset, and only report our results based on these groupings:

Table 2: Summary of warnings produced by popular bug-finding tools on the various kernels that we analyzed.

	Number of Warnings			
Kernel	**cppcheck**	**flawfinder**	**RATS**	**Sparse**
Qualcomm	18	4,365	693	5,202
Samsung	22	8,173	2,244	1,726
Hauwei	34	18,132	2,301	11,230
Mediatek	168	14,230	3,730	13,771
	242	**44,900**	**8,968**	**31,929**

Mediatek:

- Amazon Echo (5.5.0.3)
- Amazon Fire HD8 (6th Generation, 5.3.2.1)
- HTC One Hima (3.10.61-g5f0fe7e)
- Sony Xperia XA (33.2.A.3.123)

Qualcomm

- HTC Desire A56 (a56uhl-3.4.0)
- LG K8 ACG (AS375)
- ASUS Zenfone 2 Laser (ZE550KL / MR5-21.40.1220.1794)

Huawei

- Huawei Venus P9 Lite (2016-03-29)

Samsung

- Samsung Galaxy S7 Edge (SM-G935F_NN)

To ensure that we had a baseline comparison for DR. CHECKER, we also analyzed these drivers using 4 popular open-source, and stable, static analysis tools (flawfinder [57], RATs [45], cppcheck [34], and Sparse [54]). We briefly describe our interactions with each below, and a summary of the number of warnings raised by each is shown in Table 2.

Flawfinder & RATs Both Flawfinder and RATs are pattern-matching-based tool used to identify potentially dangerous portions of C code. In our experience, the installation and usage of each was quite easy; they both installed without any configuration and used a simple command-line interface. However, the criteria that they used for their warnings tended to be very simplistic, missed complex bugs, and where overly general, which resulted in an extremely high number of warnings (64,823 from Flawfinder and 13,117 from RATs). For example, Flawfinder flagged a line of code with the warning, *High: fixed size local buffer*. However, after manual investigation it was clear this code was unreachable, as it was inside of an `#if 0` definition.

We also found numerous cases where the string-matching algorithm was overly general. For example, Flawfinder raised a critical warning (*[4] (shell) system*), incorrectly reporting that `system` was being invoked for the following define: `#define system_cluster(system, clusterid)`.

Table 3: Comparison of the features provided by popular bug-finding tools and DR. CHECKER, where √ indicates availability of the feature.

Feature	cppcheck	flawfinder	RATS	Sparse	DR. CHECKER
Extensible	√	-	-	-	√
Inter-procedural	-	-	-	-	√
Handles pointers	-	-	-	-	√
Kernel Specific	-	-	-	√	√
No Manual Annotations	√	√	√	-	√
Requires compilable sources	√	-	-	√	√
Sound	-	-	-	-	-
Traceable Warnings	-	-	-	√	√

Ultimately, the tools seemed reasonable for basic code review passes, and perhaps for less-security minded programs, as they do offer informational warning messages:

> *Flawfinder:* Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.
>
> *RATs:* Check buffer boundaries if calling this function in a loop and make sure you are not in danger of writing past the allocated space

Sparse Sparse was developed by Linus Torvalds and is specifically targeted to analyze kernel code. It is implemented as a compiler front end (enabled by the flag `C=2` during compilation) that raises warnings about known problems, and even allows developers to provide static type annotations (e.g., `__user` and `__kernel`). The tool was also relatively easily to use. Although, Sparse is good at finding annotation mis-matches like unsafe user pointer dereferences [16]. Its main drawback was the sheer number of warnings (64,823 in total) it generated, where most of the warnings generated were regarding non-compliance to good kernel code practices. For example, warnings like, "*warning: Using plain integer as NULL pointer*" and "*warning: symbol 'htc_smem_ram_addr' was not declared. Should it be static?*," were extremely common.

cppcheck Cppcheck was the most complicated to use of the tools that we evaluated, as it required manual identification of all of the includes, configurations, etc. in the source code. However, this knowledge of the source code structure did result in much more concise results. While the project is open-source, their analysis techniques are not well-documented. Nevertheless, it is clear that the tool can handle more complex interactions (e.g., macros, globals, and loops) than the other three. For example, in one of the raised warnings it reported an out-of-bounds index in an array lookup. Unfortunately, after manual investigation there was a guard condition protecting the array access, but this was still a much more valuable warning that those returned by other tools. It was also able to identify an interesting use of `snprintf` on overlapped objects, which exhibits undefined behavior, and appeared generally useful. It also has a configurable engine, which allows users to specify additional types of vulnerability patterns to identify. Despite this functionality, it still failed to detect any of the complex bugs that DR. CHECKER was able to help us discover.

To summarize our experience, we provide a side-by-side feature comparison of the evaluated tools and DR. CHECKER in Table 3. Note that cppcheck and DR. CHECKER where the only two with an extensible framework that can be used to add vulnerability detectors. Similarly, every tool aside from Sparse, which needs manual annotations, was more-or-less completely automated. As previously mentioned, Sparse's annotations are used to find unsafe user pointer dereferences, and while these annotations are used rigorously in the mainline kernel code, they are not always used in the vendor drivers. Moreover, typecasting is frequently used in Linux kernel making Sparse less effective. Pattern-based tools like flawfinder and RATS do not require compilable source code, which results in spurious warnings because of pre-processor directives making them unusable. Of the evaluated features, traceability of the warnings is potentially the most important for kernel bug-finding tools [26], as these warnings will ultimately be analyzed by a human. We consider a warning to be traceable if it includes all of the information required to understand how a user input can result in the warning. In DR. CHECKER, we use the debug information embedded in the LLVM bitcode to provide traceable warnings. An example of a warning produced by DR. CHECKER is as shown in Listing 7.

7.1 DR. CHECKER

The summarized results of all of the warnings that were reported by DR. CHECKER are presented in Table 4. In this table, we consider a warning as *correct* if the report and trace were in fact true (e.g., a tainted variable was be-

Table 4: Summary of the bugs identified by DR. CHECKER in various mobile kernel drivers. We list the total number of warnings raised, number correct warnings, and number of bugs identified as a result.

	Warnings per Kernel (Count / Confirmed / Bug)				
Detector	**Huawei**	**Qualcomm**	**Mediatek**	**Samsung**	**Total**
TaintedSizeDetector	62 / 62 / 5	33 / 33 / 2	155 / 153 / 6	20 / 20 / 1	270 / 268 / 14
TaintedPointerDereferenceChecker	552 / 155 / 12	264 / 264 / 3	465 / 459 / 6	479 / 423 / 4	1760 / 1301 / 25
TaintedLoopBoundDetector	75 / 56 / 4	52 / 52 / 0	73 / 73 / 1	78 / 78 / 0	278 / 259 / 5
GlobalVariableRaceDetector	324 / 184 / 38	188 / 108 / 8	548 / 420 / 5	100 / 62 / 12	1160 / 774 / 63
ImproperTaintedDataUseDetector	81 / 74 / 5	92 / 91 / 3	243 / 241 / 9	135 / 134 / 4	551 / 540 / 21
IntegerOverflowDetector	250 / 177 / 6	196 / 196 / 2	247 / 247 / 6	99 / 87 / 2	792 / 707 / 16
KernelUninitMemoryLeakDetector	9 / 7 / 5	1 / 1 / 0	8 / 5 / 5	6 / 2 / 1	24 / 15 / 11
InvalidCastDetector	96 / 13 / 2	75 / 74 / 1	9 / 9 / 0	56 / 13 / 0	236 / 109 / 3
	1,449 / 728 / 78	901 / 819 / 19	1,748 / 1,607 / 44	973 / 819 / 24	**5,071 / 3,973 / 158**

ing used by a dangerous function). All of these warnings were manually verified by the authors, and those that are marked as a *bug* were confirmed to be critical zero-day bugs, which we are currently in the process of disclosing to the appropriate vendors. In fact, 7 of the 158 identified zero-days have already been issued Common Vulnerabilities and Exposures (CVE) identifiers [6–10]. Of these, Sparse correctly identified 1, flawfinder correctly identified 3, RATs identified 1 of the same ones as flawfinder, and cppcheck failed to identify any of them. These bugs ranged from simple data leakages to arbitrary code execution within the kernel. We find these results very promising, as 3,973 out of the 5,071 were confirmed, giving us a precision of 78%, which is easily within the acceptable 30% range [14].

While the overall detection rate of DR. CHECKER is quite good (e.g., KernelUninitMemoryLeakDetector raised 24 warnings, which resulted in 11 zero-day bugs), there a few notable *lessons learned*. First, because our vulnerability detectors are stateless, they raise a warning for every occurrence of the vulnerable condition, which results in a lot of correlated warnings. For example, the code `i = tainted+2; j = i+1;` will raise two IntegerOverflowDetector warnings, once for each vulnerable condition. This was the main contributor to the huge gap between our confirmed warnings and the actual bugs as each bug was the result of multiple warnings. The over-reporting problem was amplified by our context-sensitive analysis. For example, if a function with a vulnerable condition is called multiple times from different contexts, DR. CHECKER will raise one warning for each context.

GlobalVariableRaceDetector suffered from numerous false positives because of granularity of the LLVM instructions. As a result, the detector would raise a warning for any access to a global variable outside of a critical section. However, there are cases where the mutex object is stored in a structure field (e.g., `mutex_lock(&global->obj)`). This results in a false positive because our detector will raise a warning on the access to the global structure, despite the fact that it is completely safe, because the field inside of it is actually a mutex.

TaintedPointerDerefenceDetectors similarly struggled with the precision of its warnings. For example, on Huawei drivers (row 2, column 1), it raised 552 warnings, yet only 155 were true positives. This was due to the over-approximation of our points-to analysis. In fact, 327 of these are attributed to only two entry points `rpmsg_hisi_write` and `hifi_misc_ioctl`, where our analysis over-approximated a single field that was then repeatedly used in the function. A similar case happened for entry point `sc_v4l2_s_crop` in Samsung, which resulted in 21 false warnings. The same over-approximation of points-to affected InvalidCastDetector, with 2 entry points (`picolcd_debug_flash_read` and `picolcd_debug_flash_write`) resulting in 66 (80%) false positives in Huawei and a single entry point (`touchkey_fw_update.419`) accounting for a majority of the false positives in Samsung. IntegerOverflowDetector also suffered from over-approximation at times, with 30 false warnings in a single entry point `hifi_misc_ioctl` for Hauwei.

One notable takeaway from our evaluation was that while we expected to find numerous integer overflow bugs, we found them to be far more prevalent in 32 bit architectures than 64 bites, which is contrary to previously held beliefs [58]. Additionally, DR. CHECKER was able to correctly identify the critical class of Boomerang [33] bugs that were recently discovered.

7.2 Soundy Assumptions

DR. CHECKER in total analyzed 1207 entry points and 90% of the entry points took less than 100 seconds to complete. DR. CHECKER's practicality and scalability are made possible by our *soundy* assumptions. Specifically, not analyzing core kernel functions and not waiting for loops to converge to a fixed-point. In this section, we evaluate how these assumptions affected both

Table 5: Runtime comparison of 100 randomly selected entry points with our analysis implemented a "sound" analysis (*Sound*), a soundy analysis, without analyzing kernel functions (*No API*), and a soundy analysis without kernel functions or fixed-point loop analysis (DR. CHECKER).

	Runtime (seconds)			
Analysis	**Avg.**	**Min.**	**Max.**	**St. Dev.**
Sound*	175.823	0.012	2261.468	527.244
No API	110.409	0.016	2996.036	455.325
DR. CHECKER	35.320	0.008	978.300	146.238

* Only 18/100 sound analyses completed successfully.

our precision (i.e., practicality) and runtime (i.e., scalability). This analysis was done by randomly selecting 25 entry points from each of our codebases (i.e., Huawei, Qualcomm, Mediatek, and Samsung), resulting 100 randomly selected driver entry points. We then removed our two soundy assumptions, resulting in a "sound" analysis, and ran our analysis again.

Kernel Functions Our assumption that all kernel functions are bug free and correctly implemented is critical for the efficacy of DR. CHECKER for two reasons. First, the state explosion that results from analyzing all of the core kernel code makes much of our analysis computationally infeasible. Second, as previously mentioned, compiling the Linux kernel for ARM with LLVM is still an ongoing project, and thus would require a significant engineering effort [52]. In fact, in our evaluation we compiled the 100 randomly chosen entry with best-effort compilation using LLVM, where we created a consolidated bitcode file for each entry point with all the required kernel API functions, caveat those that LLVM failed to compile. We ran our "sound" analysis with these compiled API functions and evaluated all loops until both our points-to and taint analysis reached a fixed point, and increased our timeout window to *four hours* per entry point. Even with the potentially missing kernel API function definitions, only 18 of these 100 entry points finished within the 4 hours. The first row (*Sound*) in Table 5 shows the distribution of time over these 18 entry points. Moreover, these 18 entry points produced 63 warnings and took a total of 52 minutes to evaluate, compared to 9 warnings and less than 1 minute of evaluation time using our soundy analysis.

Fixed-point Loop Analysis Since we were unable to truly evaluate a *sound* analysis, we also evaluated our second assumption (i.e., using a reach-def loop analysis instead of a fixed-point analysis) in isolation to examine its impact on DR. CHECKER. In this experiment, we ignored the kernel API functions (i.e., assume correct implementation), but evaluated all loops until they reached a fixed point on the same 100 entry points. In this case, all of the entry points were successfully analyzed within our four hour timeout window. The second row (*No API*) in Table 5 shows the distribution of evaluation times across these entry points. Note that this approach takes 3× more time than the DR. CHECKER approach to analyze an entry point on average. Similarly, our soundy analysis returned significantly fewer warnings, 210 compared to the 474 warnings that were raised by this approach.

A summary of the execution times (i.e., sound, fixed-point loops, and DR. CHECKER) can be found in Table 5, which shows that ignoring kernel API functions is the main contributor of the DR. CHECKER's scalability. This is not surprising because almost all the kernel drivers themselves are written as kernel modules [2], which are small (7.3K lines of code on average in the analyzed kernels) and self-contained.

8 Discussion

Although DR. CHECKER is designed for Linux kernel drivers, the underlying techniques are generic enough to be applied to other code bases. Specifically, as shown in Section 7.1, ignoring external API functions (i.e., kernel functions) is the major contributor to the feasibility of DR. CHECKER on the kernel drivers. DR. CHECKER in principle can be applied to any code base, which is modular and has well-defined entry points (e.g., ImageMagick [1]). While our techniques are portable, some engineering effort is likely needed to change the detectors and setup the LLVM build environment. Specifically, to apply DR. CHECKER, one needs to:

1. Identify the source files of the module, and compile them in to a consolidated bitcode file.
2. Identify the function names, which will serve as entry points.
3. Identify how the arguments to these functions are tainted.

We provided more in-depth documentation of how this would be done in practice on our website.

9 Related Work

Zakharov *et al.* [65] discuss many of the existing tools and propose a pluggable interface for future static-analysis techniques, many of which are employed in DR. CHECKER. A few different works looked into the API-misuse problem in kernel drivers. APISan [64] is

Listing 7: Example of output from DR. CHECKER

```
At Calling Context:
    %call25 = call i64 @ged_dispatch(%struct._GED_BRIDGE_PACKAGE* %sBridgePackageKM), !dbg !27823,
     src line:187 drivers/misc/mediatek/gpu/ged/src/ged_main.c
Found:1 warning.

Warning:1
Potential vulnerability detected by:IntegerOverflowDetector :
 Potential overflow, using tainted value in a binary operation at:
%add = add i32 %2, %3, !dbg !27792,
    src line:101 drivers/misc/mediatek/gpu/ged/src/ged_main.c, Func:ged_dispatch
 Taint Trace:
%call2 = call i64 @__copy_from_user(i8* %pvTo, i8* %pvFrom, i64 %ulBytes), !dbg !27796,
    src line:43 drivers/misc/mediatek/gpu/ged/src/ged_base.c, Func:ged_copy_from_user
%2 = load i32, i32* %i32InBufferSize3, align 8, !dbg !27790,
    src line:101 drivers/misc/mediatek/gpu/ged/src/ged_main.c, Func:ged_dispatch
```

a symbolic-execution-based approach, and Static Driver Verifier (SDV) [12] similarly identified API-misuse using static data-flow analysis. However, these techniques are contrary to DR. CHECKER, as we explicitly assume that the kernel APIs are implemented properly.

SymDrive [43] uses symbolic execution to verify properties of kernel drivers. However, it requires developers to annotate their code and relies heavily on the bug finder to implement proper *checkers*. Johnson *et al.* [28] proposed a sound CQUAL-based [24] tool, which is context-sensitive, field-sensitive, and precise taint-based analysis; however, this tool also requires user annotations of the source code, which DR. CHECKER does not.

KINT [56] uses taint analysis to find integer errors in the kernel. While KINT is sound, their techniques are specialized to integer errors, whereas DR. CHECKER attempts to find general input validation errors by compromising soundness.

Linux Driver Verification (LDV) [36] is a tool based on BLAST [27] that offers precise pointer analysis; however, it is still a model-checker-based tool, whereas we built our analysis on well-known static analysis techniques. Yamaguchi *et al.* have done a significant amount of work in this area, based on Joern [59–62], where they use static analysis to parse source code into novel data structures and find known vulnerable signatures. However, their tool is similar to a pattern-matching model-checking type approach, whereas we are performing general taint and points-to analysis with pluggable vulnerability detectors. VCCFinder [41] also used a similar pattern-matching approach, but automatically constructed their signatures by training on previously known vulnerabilities to create models that could be used to detect future bugs.

MECA [63] is a static-analysis framework, capable of taint analysis, that will report violations based on user annotations in the source code, and similarly aims to reduce false positives by sacrificing soundness. ESP [22] is also capable of fully path-sensitive partial analysis using "property simulation," wherein they combine data-flow analysis with a property graph. However, this approach is not as robust as our more general approach.

Boyd-Wickizer *et al.* [15] proposed a potential defense against driver vulnerabilities that leverages x86 hardware features; however, these are unlikely to be easily ported to ARM-based mobile devices. *Nooks* [49] is a similar defense; however, this too has been neglected in both the mainline and mobile deployments thus far, due to similar hardware constraints.

10 Conclusion

We have presented DR. CHECKER, a fully-automated static analysis bug-finding tool for Linux kernels that is capable of general context-, path-, and flow-sensitive points-to and taint analysis. DR. CHECKER is based on well-known static analysis techniques and employs a *soundy* analysis, which enables it to return precise results, without completely sacrificing soundness. We have implemented DR. CHECKER in a modular way, which enables both analyses and bug detectors to be easily adapted for real-world bug finding. In fact, during the writing of this paper, we identified a new class of bugs and were able to quickly augment DR. CHECKER to identify them, which resulted in the discovery 63 zero-day bugs. In total, DR. CHECKER discovered 158 previously *undiscovered* zero-day bugs in nine popular mobile Linux kernels. All of the details and disclosures for these bugs can be found online at `github.com/ucsb-seclab/dr_checker`. While these results are promising, DR. CHECKER still suffers from over-approximation as a result of being *soundy*, and we have identified areas for future work. Nevertheless, we feel that DR. CHECKER exhibits the importance of analyzing Linux kernel drivers and provides a useful framework for adequately handling this complex code.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Stelios Sidiroglou-Douskos for their valuable comments and input to improve our paper. This material is based on research sponsored by the Office of Naval Research under grant number N00014-15-1-2948 and by DARPA under agreement number FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

This work is also sponsored by a gift from Google's Anti-Abuse group.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Imagemagick: Convert different image formats. `https://github.com/ImageMagick/ImageMagick`. Accessed: 2017-05-26.

[2] Kernel modules. `http://tldp.org/LDP/lkmpg/2.6/html/x427.html`. Accessed: 2017-05-26.

[3] The linux watchdog timer driver core kernel api. `https://www.kernel.org/doc/Documentation/watchdog/watchdog-kernel-api.txt`. Accessed: 2017-02-14.

[4] CVE-2016-2068. Available from MITRE, CVE-ID CVE-2016-2068., 2016.

[5] CVE-2016-5195. Available from MITRE, CVE-ID CVE-2016-5195., May 2016.

[6] CVE-2016-8433. Available from MITRE, CVE-ID CVE-2016-8433., May 2016.

[7] CVE-2016-8448. Available from MITRE, CVE-ID CVE-2016-8448., May 2016.

[8] CVE-2016-8470. Available from MITRE, CVE-ID CVE-2016-8470., May 2016.

[9] CVE-2016-8471. Available from MITRE, CVE-ID CVE-2016-8471., May 2016.

[10] CVE-2016-8472. Available from MITRE, CVE-ID CVE-2016-8472., May 2016.

[11] AUBERT, J., AND TUSET, D. c2xml. `http://c2xml.sourceforge.net/`.

[12] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review 40*, 4 (2006), 73–85.

[13] BALL, T., AND RAJAMANI, S. K. The slam project: Debugging system software via static analysis. In *Proceedings of the 2002 ACM Symposium on Principles of Programming Languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 1–3.

[14] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.

[15] BOYD-WICKIZER, S., AND ZELDOVICH, N. Tolerating malicious device drivers in linux. In *Proceedings of the 2010 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 9–9.

[16] BUGRARA, S., AND AIKEN, A. Verifying the safety of user pointer dereferences. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 325–338.

[17] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2011 Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, ACM, pp. 5:1–5:5.

[18] CHEN, H., AND WAGNER, D. Mops: An infrastructure for examining security properties of software. In *Proceedings of the 2002 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2002), CCS '02, ACM, pp. 235–244.

[19] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *Proceedings of the 2001 ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 73–88.

[20] CORBET, J., RUBINI, A., AND KROAH-HARTMAN, G. *Linux Device Drivers: Where the Kernel Meets the Hardware*. " O'Reilly Media, Inc.", 2005.

[21] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. An efficient method of computing static single assignment form. In *Proceedings of the 1989 ACM Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), POPL '89, ACM, pp. 25–35.

[22] DAS, M., LERNER, S., AND SEIGLE, M. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 57–68.

[23] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 51–62.

[24] FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 1–12.

[25] GE, X., VIJAYAKUMAR, H., AND JAEGER, T. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747* (2014).

[26] GUO, P. J., AND ENGLER, D. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the 2009 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIXATC'09, USENIX Association, pp. 22–22.

[27] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Software verification with blast. In *Proceedings of the 2003 International Conference on Model Checking Software* (Berlin, Heidelberg, 2003), SPIN'03, Springer-Verlag, pp. 235–239.

[28] JOHNSON, R., AND WAGNER, D. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 2004 USENIX Conference on Security* (Berkeley, CA, USA, 2004), SEC'04, USENIX Association, pp. 9–9.

[29] KIRAT, D., VIGNA, G., AND KRUEGEL, C. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 2014 USENIX Conference on Security* (Berkeley, CA, USA, 2014), SEC'14, USENIX Association, pp. 287–301.

[30] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[31] LIVSHITZ, B. Soundness is not even necessary for most modern analysis applications, however, as many. *Communications of the ACM 58*, 2 (2015).

[32] LU, K., SONG, C., KIM, T., AND LEE, W. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 920–932.

[33] MACHIRY, A., GUSTAFSON, E., SPENSKY, C., SALLS, C., STEPHENS, N., WANG, R., BIANCHI, A., CHOE, Y. R., KRUEGEL, C., AND VIGNA, G. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)* (2017), Internet Society.

[34] MARJAMÄKI, D. Cppcheck: a tool for static c/c++ code analysis. `http://cppcheck.sourceforge.net/`, December 2016.

[35] MOCHEL, P., AND MURPHY, M. sysfs - _The_ filesystem for exporting kernel objects. `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`.

[36] MUTILIN, V., NOVIKOV, E., STRAKH AV, K. A., AND SHVED, P. Linux driver verification [linux driver verification architecture]. *Trudy ISP RN [The Proceedings of ISP RAS] 20* (2011), 163–187.

[37] NEIRA-AYUSO, P., GASCA, R. M., AND LEFEVRE, L. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience 40*, 9 (2010), 797–810.

[38] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of program analysis*. Springer, 2015.

[39] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in linux: Ten years later. In *Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS'11, ACM, pp. 305–318.

[40] PEIRÓ, S., MUÑOZ, M., MASMANO, M., AND CRESPO, A. Detecting stack based kernel information leaks. In *Proceedings of the 2014 International Joint Conference SOCO'14-CISIS'14-ICEUTE'14* (2014), Springer, pp. 321–331.

[41] PERL, H., DECHAND, S., SMITH, M., ARP, D., YAMAGUCHI, F., RIECK, K., FAHL, S., AND ACAR, Y. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 2015 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 426–437.

[42] QUINTAO PEREIRA, F. M., RODRIGUES, R. E., AND SPERLE CAMPOS, V. H. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2013), CGO '13, IEEE Computer Society, pp. 1–11.

[43] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. Symdrive: Testing drivers without devices. In *Proceedings of the 2012 USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 279–292.

[44] SALZMAN, P. J., BURIAN, M., AND POMERANTZ, O. Hello World (part 3): The __init and __exit Macros. `http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html#AEN245`, May 2007.

[45] SECURE SOFTWARE, I. Rats - rough auditing tool for security. `https://github.com/andrew-d/rough-auditing-tool-for-security`, December 2013.

[46] SPENSKY, C., HU, H., AND LEACH, K. Lo-phi: Low-observable physical host instrumentation for malware analysis. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)* (2016), Internet Society.

[47] SPENSKY, C., STEWART, J., YERUKHIMOVICH, A., SHAY, R., TRACHTENBERG, A., HOUSLEY, R., AND CUNNINGHAM, R. K. SoK: Privacy on Mobile Devices–It's Complicated. *Proceedings on Privacy Enhancing Technologies 2016*, 3 (2016), 96–116.

[48] STOEP, J. V. Android: protecting the kernel. *Linux Securit Summit* (August 2016).

[49] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 2003 ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 207–222.

[50] TARJAN, R. Depth-first search and linear graph algorithms. *SIAM journal on computing 1*, 2 (1972), 146–160.

[51] THE CLANG PROJECT. clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[52] THE LINUX FOUNDATION. LLVMLinux Project Overview. `http://llvm.linuxfoundation.org/index.php/Main_Page`.

[53] THE LLVM PROJECT. The Often Misunderstood GEP Instruction. `http://llvm.org/docs/GetElementPtr.html`.

[54] TORVALDS, L., TRIPLETT, J., AND LI, C. Sparse–a semantic parser for c. *see http://sparse. wiki. kernel. org* (2007).

[55] VIEGA, J., BLOCH, J. T., KOHNO, Y., AND MCGRAW, G. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 2000 Annual Computer Security Applications Conference* (Washington, DC, USA, 2000), ACSAC '00, IEEE Computer Society, pp. 257–.

[56] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Improving integer security for systems with kint. In *Proceedings of the 2012 USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 163–177.

[57] WHEELER, D. A. Flawfinder, 2011.

[58] WRESSNEGGER, C., YAMAGUCHI, F., MAIER, A., AND RIECK, K. Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 541–552.

[59] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 590–604.

[60] YAMAGUCHI, F., LOTTMANN, M., AND RIECK, K. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 2012 Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 359–368.

[61] YAMAGUCHI, F., MAIER, A., GASCON, H., AND RIECK, K. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2015), SP '15, IEEE Computer Society, pp. 797–812.

[62] YAMAGUCHI, F., WRESSNEGGER, C., GASCON, H., AND RIECK, K. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 499–510.

[63] YANG, J., KREMENEK, T., XIE, Y., AND ENGLER, D. Meca: An extensible, expressive system and language for statically checking security properties. In *Proceedings of the 2003 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 321–334.

[64] YUN, I., MIN, C., SI, X., JANG, Y., KIM, T., AND NAIK, M. Apisan: Sanitizing api usages through semantic cross-checking. In *Proceedings of the 2016 USENIX Conference on Security*, SEC'16, USENIX Association, pp. 363–378.

[65] ZAKHAROV, I. S., MANDRYKIN, M. U., MUTILIN, V. S., NOVIKOV, E. M., PETRENKO, A. K., AND KHOROSHILOV, A. V. Configurable toolset for static verification of operating systems kernel modules. *Program. Comput. Softw. 41*, 1 (Jan. 2015), 49–64.

[66] ZHANG, H., LI, X.-H., LIU, B., AND QIAN, X. The video device driver programming and profiting based on v4l2 [j]. *Computer Knowledge and Technology 15* (2010), 062.