K-Miner: Uncovering Memory Corruption in Linux

David Gens,* Simon Schmitt,* Lucas Davi,† Ahmad-Reza Sadeghi*
*CYSEC/Technische Universität Darmstadt, Germany.
{david.gens,simon.schmitt,ahmad.sadeghi}@trust.tu-darmstadt.de
†University of Duisburg-Essen, Germany. lucas.davi@uni-due.de

***Abstract*—Operating system kernels are appealing attack targets: compromising the kernel usually allows attackers to bypass all deployed security mechanisms and take control over the entire system. Commodity kernels, like Linux, are written in low-level programming languages that offer only limited type and memory-safety guarantees, enabling adversaries to launch sophisticated run-time attacks against the kernel by exploiting memory-corruption vulnerabilities.**

Many defenses have been proposed to protect operating systems at run time, such as control-flow integrity (CFI). However, the goal of these run-time monitors is to prevent exploitation as a symptom of memory corruption, rather than eliminating the underlying root cause, i.e., bugs in the kernel code. While finding bugs can be automated, e.g., using static analysis, all existing approaches are limited to local, intra-procedural checks, and face severe scalability challenges due to the large kernel code base. Consequently, there currently exist no tools for conducting global static analysis of operating system kernels.

In this paper, we present *K-Miner*, a new framework to efficiently analyze large, commodity operating system kernels like Linux. Our novel approach exploits the highly standardized interface structure of the kernel code to enable scalable pointer analysis and conduct global, context-sensitive analysis. Through our inter-procedural analysis we show that K-Miner systematically and reliably uncovers several different classes of memory-corruption vulnerabilities, such as dangling pointers, use-after-free, double-free, and double-lock vulnerabilities. We thoroughly evaluate our extensible analysis framework, which leverages the popular and widely used LLVM compiler suite, for the current Linux kernel and demonstrate its effectiveness by reporting several memory-corruption vulnerabilities.

I. Introduction

Operating system kernels form the foundation of practically all modern software platforms. The kernel features many important services and provides the interfaces towards user applications. It is usually isolated from these applications through hardware mechanisms such as memory protection and different privilege levels in the processor. However, memory-corruption vulnerabilities in the kernel code open up the possibility for unprivileged users to subvert control flow or data structures and take control over the entire system [32], [70], [72], [47]. For this reason, many defenses have been proposed in the past [25], [37], [49], [16], [57], [5], [23], [63], [20]. These defenses are designed specifically for run-time protection of operating system kernels. Their goal is to provide countermeasures and secure the kernel against attacks exploiting memory corruption. Most of these approaches can be loosely categorized as run-time monitors [49], [16], [57], [5], [23], [63], [19].

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23326
www.ndss-symposium.org

Run-time Monitors vs. Compile-time Verification. Typically, adversaries are modeled according to their capabilities, and reference monitors are then designed to defend against a specific class of attacks. For instance, control-flow integrity (CFI) is tailored towards control-flow hijacking attacks. However, CFI is not designed to protect against data-only adversaries resulting in a protection gap that allows for crucial attacks despite the presence of run-time monitors, such as CFI, in the kernel [11], [32], [70], [20]. Thus, a combination of many different defenses is required to protect the kernel against multiple classes of adversaries. Consequently, commodity operating systems will remain vulnerable to new types of software attacks as long as memory-corruption vulnerabilities are present in the code [60].

An alternative approach to employing run-time monitors is to ensure the absence of memory-corruption vulnerabilities by analyzing the system before deployment. This was shown to be feasible for small microkernels with less than 10,000 lines of code [6], [44], [64], by building a formal model of the entire kernel and (manually) proving the correctness of the implementation with respect to this model. The invariants that hold for the formal model then also hold for the implementation. However, the formal correctness approach is impractical for commodity monolithic kernels due to their size and extensive use of machine-level code [43], which provides no safety guarantees. Even for small microkernels formulating such a model and proving its correctness requires more than 10 person years of work [44], [6]. While dynamic analyses are used to detect vulnerabilities in OS kernels rather successfully [34], [35], [22], static approaches have a major advantage: sound static analysis safely over-approximates program execution, allowing for strong statements in the case of negative analysis results. In particular, if no report is generated for a certain code path by a sound analysis, one can assert that no memory-corruption vulnerability is present. Hence, static analysis is also a practical and pragmatic alternative to formal verification, as it is able to offer similar assurances for real-world software by means of automated compile-time checks [15].

Static analysis of commodity kernels. However, static analysis faces severe scalability challenges, and hence, all analysis frameworks for kernel code are limited to *intra-procedural* analysis, i.e., local checks per function.

In particular, there are five popular analysis frameworks targeting Linux: Coccinelle [52], Smatch [9], TypeChef [38], APISAN [74], and EBA [1]. None of these support *inter-procedural* data-flow analyses, which are required to conservatively approximate program behavior, and reliably uncover memory corruption caused by global pointer relationships. The main reason why precise data-flow analysis for kernel code represents a huge challenge for all existing approaches, is the huge size and complexity of its monolithic code base: currently, Linux comprises over 24 million lines of code [14]. Just compiling a common distribution kernel takes several hours, even on top-of-the-line hardware. While some of the existing tools allow for the static analysis of kernel code, these are conceptually limited to local intra-procedural (i.e., per-function) or simple file-based analysis. This limitation is due to the fact that the number of possible paths grows exponentially with the code size, and hence, static analysis approaches face severe scalability problems [29], [28], [65]. At the same time, analysis methods have to take all paths and states into account to remain sound, and hence, pruning or skipping certain parts of the code would lead to unreliable results. This is why the resource requirements for conducting such analyses in the Linux kernel quickly outgrows any realistic thresholds. As a result, global and inter-procedural analyses, which are required to uncover memory corruption reliably, remain out of reach of these existing approaches.

Goals and Contribution.
In this paper, we propose to exploit a distinct and unique property of kernel software: its interface to user space is highly standardized [33]. Our idea is to partition the kernel code along separate execution paths using the system call interface as a starting point. We show that this significantly reduces the number of relevant paths, allowing us to conduct even complex, inter-procedural data-flow analysis in the kernel. To this end, we present the design and implementation of *K-Miner*, the first static analysis framework that enables complex data-flow analysis for Linux to reliably detect vulnerabilities in kernel code.

Partitioning the kernel code comes with a number of challenges, such as the frequent reuse of global data structures, the synchronization between the per-system call and global memory states (contexts), and complicated and deeply nested aliasing relationships of pointers. As we will show, our framework tackles all of these challenges, providing a number of different analysis passes that analyze system calls simultaneously, and reporting a number of real-world vulnerabilities.

Further, scalable static analysis designed for user space programs cannot simply be applied in kernel setting: data-flow analysis expects an initial state from which analysis passes propagate value flows, which is naturally satisfied by a program's `main` function in user space. K-Miner is tailored towards this requirement and enables data-flow analysis in the kernel setting.

To summarize our contributions are as follows:

- **Enable global static analysis for kernel code:** we present *K-Miner*, a novel approach to conduct global static analyses of the Linux kernel. Our framework allows to systematically analyze the kernel's user-space interface and detect possible memory corruption. To enable precise inter-procedural static analysis of modern OS kernels we tackle a number of challenges, such as dealing with the large code base, complex inter-dependencies, and aliasing relationships between global kernel pointers and memory objects.
- **Prototype framework implementation:** we provide multiple analyses for finding classes of vulnerabilities in the Linux kernel that are typically exploited, and demonstrate their effectiveness in analyzing many different kernel versions, using different configurations. Our presented framework is extensible and adding additional analysis passes is straightforward. K-Miner includes a web-based user interface to ease reporting and collaboration. It also provides extensive graph-based analysis overviews and statistics on the number of alerts and performance. We release our implementation of K-Miner as an open source project [24] that is built on top of LLVM [46].
- **Extensive evaluation:** we rigorously evaluate our static analysis framework implemenation for the Linux kernel by applying it to all system calls across many different Linux versions and highlight the effectiveness of our framework through detailed reports and statistics. We demonstrate the importance of automated and scalable analysis of commodity kernel code by reliably uncovering several known memory-corruption vulnerabilities, which previously required manual inspection of the code, and were used to conduct real-world kernel exploits against dissidents and activists [50], [62]. Using K-Miner these bug classes can now be found automatically through our precise and reliable static analysis passes. We reported two use-after-return vulnerabilities that K-Miner uncovered in the kernel.

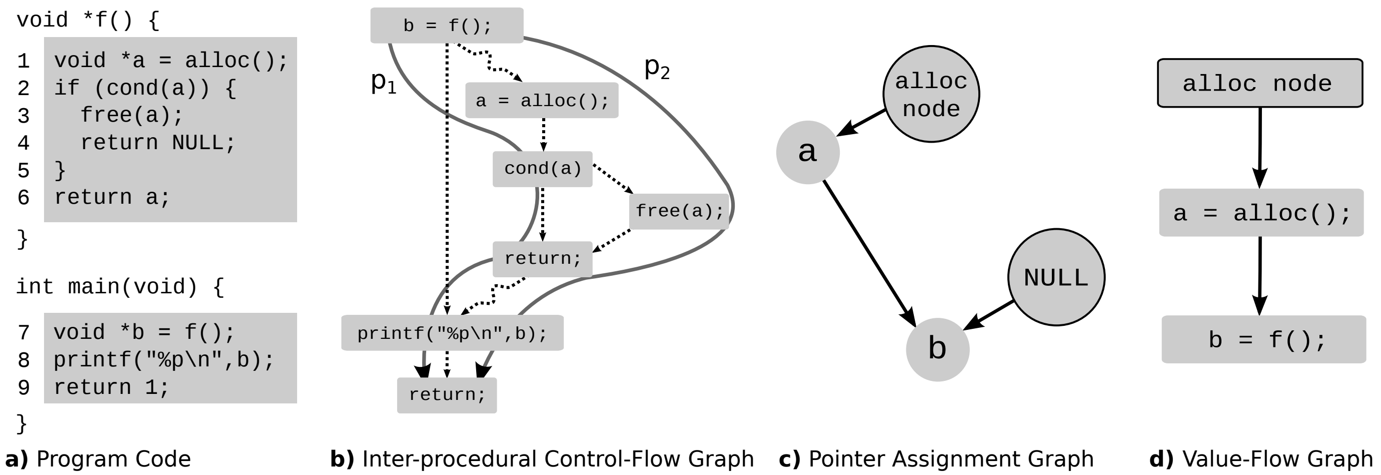
II. Background

In this section we explain the concepts behind static data-flow analysis and present a classification of memory-corruption vulnerabilities.

A. Data-Flow Analysis

The general idea of static analysis is to take a program and a list of pre-compiled properties as input, and find all the paths for which a given property is true. Examples of such properties are liveness analysis, dead-code analysis, typestate analysis, or nullness analysis [41]. For instance, a nullness analysis for the program **a)** in Figure 1 could be answered by looking at its pointer-assignment graph (PAG) depicted in **c)**: since there is a path in which variable `b` is assigned a `NULL` value (`b` points to `NULL` in the PAG) a report will be issued. Another commonly used data structure is the inter-procedural control-flow graph (ICFG) in **b)** — limited to the procedures `main` and `f` for brevity — which propagates control flow globally. This can be used to conduct path-sensitive analysis. Finally, taint and source-sink analysis may track individual memory objects through their associated value-flow graph (VFG) in **d)**.

Static analysis for tracking individual values in a program is called *data-flow analysis*. Most data-flow analysis approaches follow a general concept, or framework, to analyze programs systematically. The naive approach is to enumerate all possible program paths and test each graph for a given property. This is commonly referred to as the *Meet Over all Paths* (MOP). In Figure 1, the MOP would be calculated by testing a



```
void *f() {
void *a = alloc();
if (cond(a)) {
  free(a);
  return NULL;
}
return a;
}

int main(void) {
void *b = f();
printf("%p\n",b);
return 1;
}
```

a) Program Code **b)** Inter-procedural Control-Flow Graph **c)** Pointer Assignment Graph **d)** Value-Flow Graph

Figure 1: Data-flow analyses utilize graphs to reason about program behavior at compile time.

property against the two alternative program paths p_1 and p_2. Unfortunately, in the general case the MOP solution was shown to be undecidable by reduction to the post correspondence problem [36].

However, the MOP can be approximated through a so-called *Monotone Framework*, which is a set of mathematically defined objects and rules to analyze program behavior. At the heart of the monotone framework is a *lattice*, i.e., a partial order with a unique least upper bound that must be defined over the domain of all possible values during program execution. Further, the analysis framework must specify monotone flow functions that define how program statements effect lattice elements (the monotony requirement ensures termination of the analysis). Finally, sets of domain elements (i.e., *values*) must be combined using a *merge operator*. A commonly used definition for points-to analysis is the domain of points-to sets for all pointers in a program. The flow functions then select all program statements, which potentially modify any pointer relations and specify their target transitions in the lattice. The merge operator defines how to combine the resulting points-to sets for such a transition. The notion of the monotone framework is significant for static program analysis: for any monotone framework, there exists a Maximum Fixed Point (MFP) solution, which safely approximates the MOP solution [36]. If the flow functions are distributive under the merge operator that is defined by the lattice, the MFP solution is identical to the MOP solution. The montone framework is then called a distributive framework, and data-flow analysis problems can be solved efficiently by solving a corresponding graph reachability problem [54].

B. *Memory-corruption vulnerabilities*

Memory-corruption vulnerabilities represent a vast number of security relevant bugs for operating system software (e.g., [12], [10]). Run-time attacks exploit such bugs to inject malicious code, reuse existing code with a malicious input, or corrupt integral data structures to obtain higher privileges. Memory-corruption vulnerabilities are often classified according to their root defect: integer overflows (IO), use-after-free (UAF), dangling pointers (DP), double free (DF), buffer overflow (BO), missing pointer checks (MPC), uninitialized data (UD), type errors (TE), or synchronization errors (SE) are commonly listed classes of memory corruption [11], [60]. Any instance of memory corruption leaves the program vulnerable to run-time attacks: each class represents a violation of well-defined program behavior as specified by the programming-language standard or the compiler, and hence, the violating program can exhibit arbitrary behavior at run time. For this reason an adversary with knowledge about any such vulnerability can exploit the program by deliberately triggering the error to achieve unintended, malicious behavior.

For an operating system, the main interface which exposes kernel code to a user space adversary are system calls [61]. In our approach we combine different data-flow analysis passes for the classes listed above to report potential bugs in kernel code, which are accessible to a user space program through the system call interface. Since memory-corruption vulnerabilities account for many real-world exploits [60], we focus on reporting instances of dangling pointers (DP), user-after-free (UAF), and double free (DF) in our proof-of-concept implementation. For instance, dangling-pointer vulnerabilities occur when a memory address is assigned to a pointer variable, and the memory belonging to that address subsequently becomes unavailable, or invalid. For heap allocations this can happen, e.g., when a memory location is freed but the pointer is still accessible. For stack-based allocations this happens when the stack frame containing the allocated object becomes invalid, e.g., due to a nested return statement in or below the scope of the allocation. Our framework is extensible such that new analyses passes can be integrated to search for additional vulnerability classes (cf., Section VI).

III. K-Miner

In this section, we explain our threat model, introduce the high-level design of K-Miner, and elaborate on challenges to enable precise, inter-procedural static analysis of complex, real-world kernels.

A. Goals and assumptions

With K-Miner we aim to identify and report potential memory-corruption bugs in the kernel's user-space interface, so that developers can fix them before shipping code that includes such vulnerabilities. Regarding potential malicious processes at run time we make the following standard assumptions:

- The attacker has control over a user-space process and can issue all system calls to attack the kernel through the subverted process.
- The operating system is isolated from user processes, e.g., through virtual memory and different privilege levels. Common platforms like x86 and ARM meet this requirement.
- An adversary cannot insert malicious code into the kernel through modules, because modern operating systems require kernel modules to be cryptographically signed [45], [48], [3].
- K-Miner should reliably report memory-corruption vulnerabilities that can be triggered by a malicious process.

Our assumptions force the attacker to exploit a memory-corruption vulnerability in the kernel code to gain kernel privileges through a purely software-based attack. The goal of K-Miner is to systematically scan the system call interface for these vulnerabilities.

Since real-world adversaries are not limited to software vulnerabilities, it is important to note that even with a completely verified kernel (e.g., seL4) hardware attacks such as rowhammer [42], [55] still pose a serious threat to the integrity of the kernel. However, for our work we consider hardware implementation defects to be an orthogonal problem [7].

B. Overview

K-Miner is a static analysis framework for commodity operating system kernels. We provide a high-level overview in Figure 2.

Our framework builds on top of the existing compiler suite LLVM. The compiler (cf., step ①) receives two inputs. First, a configuration file, which contains a list of selected kernel features. This configuration file enables the user to select or deselect individual kernel features. When a feature is disabled, its code is not included in the implementation. Hence, an analysis result is only valid for a specific pair of kernel code and configuration file. Second, the compiler suite parses the kernel code according to the configuration. It syntactically checks the code and builds an abstract syntax tree (AST). The compiler then internally transforms the AST into a so-called intermediate representation (IR), which essentially represents an abstract, hypothetical machine model. The IR is also used for analyzing and optimizing kernel code through a series of transformation passes.

In step ②, the compiler suite passes the IR of the kernel as an input to K-Miner, which starts to statically check the code by going through the list of all system calls. For every system call, K-Miner generates a call graph (CG), a value-flow graph (VFG), a pointer-analysis graph (PAG), and several other internal data structures by taking the entry point of the system call function as a starting point. Additionally, we compute a list of all globally allocated kernel objects, which are reachable by any single system call. Once these data structures are generated, K-Miner can start the actual static analysis passes. There are individual passes for different types of vulnerabilities, e.g., dangling-pointer, use-after-free, double-free, and double-lock errors. All of these passes analyze the control flow of a specific system call at a time, utilizing the previously generated data structures. The passes are implemented as context-sensitive value-flow analyses: they track inter-procedural context information by taking the control flow of the given system call into account and descend in the call graph.

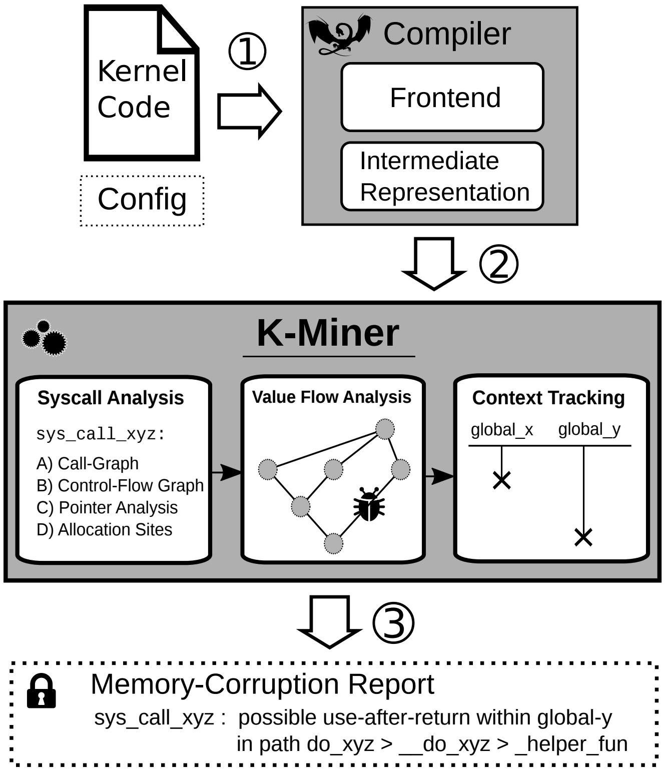


Figure 2: Overview of the different components of K-Miner.

If a potential memory-corruption bug has been detected, K-Miner generates a report, containing all relevant information (the affected kernel version, configuration file, system call, program path, and object) in step ③.

C. Uncovering Memory Corruption

The goal of K-Miner is to systematically scan the kernel's interface for different classes of memory-corruption vulnerabilities using multiple analysis passes, each tailored to find a specific class of vulnerability. The individual analysis pass utilizes data structures related to the targeted vulnerability class to check if certain conditions hold true. Reasoning about memory and pointers is essential for analyzing the behavior of the kernel with respect to memory-corruption vulnerabilities, hence, the data base for all memory objects (called *global context*) and the pointer-analysis graph represent the foundation for many analysis passes. Individual memory objects are instantiated at allocation sites throughout the entire kernel and the variables potentially pointing to them are tracked per system call using the PAG. Forward analysis then reasons about the past behaviour of an individual memory location, whereas a backward analysis determines future behaviour (since a forward analysis processes past code constructs before processing future code and vice versa).

We can also combine such analysis passes, for instance, to find double-free vulnerabilities: first, we determine sources and sinks for memory objects, i.e., allocation sites and the corresponding free functions respectively. We then process the VFG in the forward direction for every allocation site to determine reachable sinks. Second, we reconstruct the resulting paths for source-sink pairs in the execution by following sinks in the backward direction. Finally, we analyze the forward paths again to check for additional sinks. Since any path containing more than one sink will report a duplicate de-allocation this approach suffers from a high number of false positives. For this reason, we determine if the first de-allocation invocation *dominates* (i.e., is executed in every path leading to) the second de-allocation invocation in the validation phase.

In similar vein we provide passes that are checking for conditions indicating dangling pointers, use-after-free, and double-lock errors. We provide more detailed examples for the implementation of such passes in Section IV.

D. Challenges

Creating a static analysis framework for real-world operating systems comes with a series of difficult challenges, which we briefly describe in this section. In Section IV we explain how to tackle each challenge in detail.

Global state.
Most classes of memory-corruption vulnerabilities deal with pointers, and the state or type of the objects in memory that they point to. Conducting inter-procedural pointer analyses poses a difficult challenge regarding efficiency. Because inter-procedural analysis allows for global state, local pointer accesses may have non-local effects due to aliasing. Since our analyses are also flow-sensitive, these aliasing relationships are not always static, but can also be updated while traversing the control-flow graph. To enable complex global analyses, we make use of sparse program representations: we only take value flows into account that relate to the currently analyzed call graph and context information.

Huge codebase.
The current Linux kernel comprises more than 24 million lines of code [14], supporting dozens of different architectures, and hundreds of drivers for external hardware. Since K-Miner leverages complex data-flow analysis, creating data structures and dependence graphs for such large amounts of program code ultimately results in an explosion of resource requirements. We therefore need to provide techniques to reduce the amount of code for individual analysis passes without omitting any code, and allowing reuse of intermediate results. By partitioning the kernel according to the system call interface, we are able to achieve significant reduction of the number of analyzed paths, while taking all the code into account, and allowing reuse of important data structures (such as the kernel context).

False positives.
False positives represent a common problem of static analysis, caused by too coarse-grained over approximation of possible program behavior. Such over approximation results in a high number of reports that cannot be handled by developers. K-Miner has to minimize the number of false positives to an absolute minimum. As the number of false positives depends greatly on the implementation of the individual analysis passes we carefully design our analyses to leverage as much information as possible to eliminate reports that require impossible cases at run time, or make too coarse-grained approximations. Moreover, we sanitize, deduplicate, and filter generated reports before displaying them for developers in a collaborative, web-based user interface.

Multiple analyses.
A comprehensive framework needs to be able to eliminate all possible causes of memory corruption. This is why K-Miner must be able to combine the results of many different analyses. Additionally, individual analyses may depend on intermediate results of each other. Hence, our framework has to be able to synchronize these with respect to the currently inspected code parts. To this end we leverage the modern pass infrastructure of LLVM to export intermediary results and partially re-import them at a later point in time.

IV. Implementation

In this section we describe our implementation of K-Miner, and how we tackle the challenges mentioned in Section III-D. Our framework builds on the compiler suite LLVM [46] and the analysis framework SVF [59]. The former provides the basic underlying data structures, simple pointer analysis, a pass-infrastructure, and a bitcode file format which associates the source language with the LLVM intermediate representation (IR). The latter comprises various additional pointer analyses and a sparse representation of a value-flow dependence graph.

Since it is possible to compile the Linux kernel with LLVM [69], we generate the required bitcode files by modifying the build process of the kernel, and link them together to generate a bitcode version of the kernel image. This image file can then be used as input for K-Miner. Figure 3 depicts the structure of our framework implementation. In particular, it consists of four analysis stages: in step ①, the LLVM-IR is passed to K-Miner as a `vmlinux` bitcode image to start a pre-analysis, which will initialize and populate the global kernel context. In step ②, this context information is used to analyze individual system calls. It is possible to run multiple analysis passes successively, i.e., our dangling pointer, use-after-free, and double-free checkers, or run each of them independently. In step ③, bug reports are sanitized through various validation techniques to reduce the number of false positives. In step ④, the sorted reports are rendered using our vulnerability reporting engine. In the following, we describe each of the steps in more detail and explain how each of them tackles the challenges identified in the previous section.

A. Global Analysis Context

The global context stored by K-Miner essentially represents a data base for all the memory objects that are modeled based

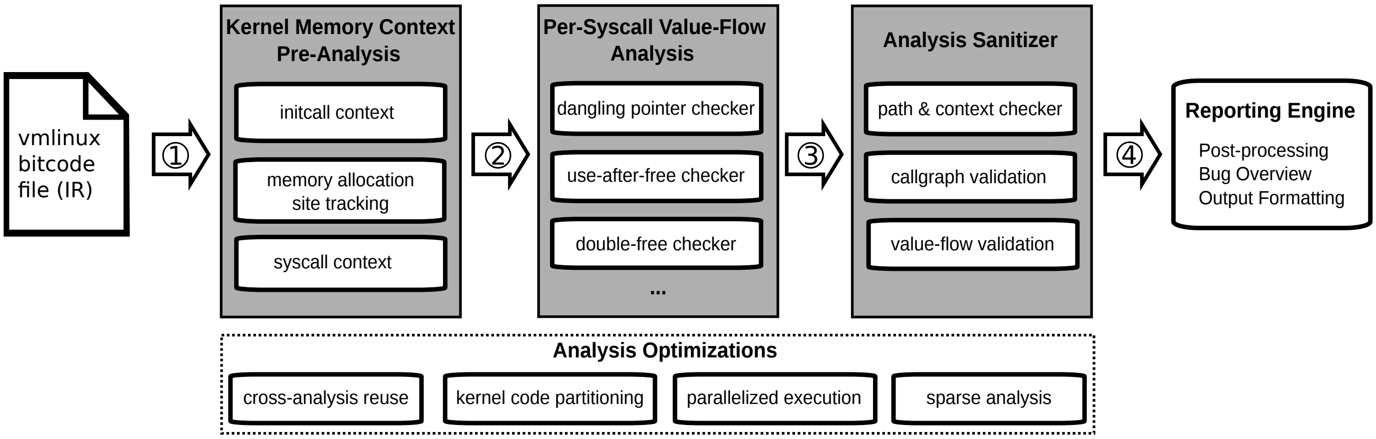


Figure 3: Overview of the K-Miner implementation: we conduct complex data-flow analysis of the Linux kernel in stages, re-using intermediate results.

on the source code. Managing global context information efficiently is a prerequisite to enable analysis of highly complex code bases such as the Linux kernel. Additionally, we have to ensure that the context is sufficiently accurate to support precise reporting in our subsequent analysis. This is why the pre-analysis steps of our framework resemble the execution model of the kernel to establish and track global kernel context information.

Initializing the Kernel Context: The kernel usually initializes its memory context at run time by populating global data structures, such as the list of tasks or virtual memory regions during early boot phase. This is done by calling a series of specific functions, called *Initcalls*. These are one-time functions which are annotated with a macro in the source files of the kernel. The macro signals the compiler to place these functions in a dedicated code segment. Functions in this segment will only be executed during boot or if a driver is loaded. Hence, most of the memory occupied by this segment can be freed once the machine finished booting [67]. To initialize the global kernel context, we populate global kernel variables by simulating the execution of these initcalls prior to launching the analyses for each system call. The resulting context information is in the order of several hundred megabytes, therefore, we export it to a file on disk and re-import it at a later stage when running individual data-flow analysis passes.

Tracking Heap Allocations: Usually, user space programs use some variant of `malloc` for allocating memory dynamically at run time. There are many different methods for allocating memory dynamically in the kernel, e.g., a slab allocator, a low-level page-based allocator, or various object caches. To enable tracking of dynamic memory objects, we have to compile a list of allocation functions which should be treated as heap allocations. Using this list K-Miner transforms the analyzed bitcode by marking all call sites of these functions as sources of heap memory. In this way kernel memory allocations can be tracked within subsequent data-flow analysis passes.

Establishing a Syscall Context: Because subsequent analysis passes will be running per system call, we establish a dedicated memory context for each of them. We do this by collecting the uses of any global variables and functions in each of the system call graphs. By cross-checking this context information against the global context, we can establish an accurate description of the memory context statically.

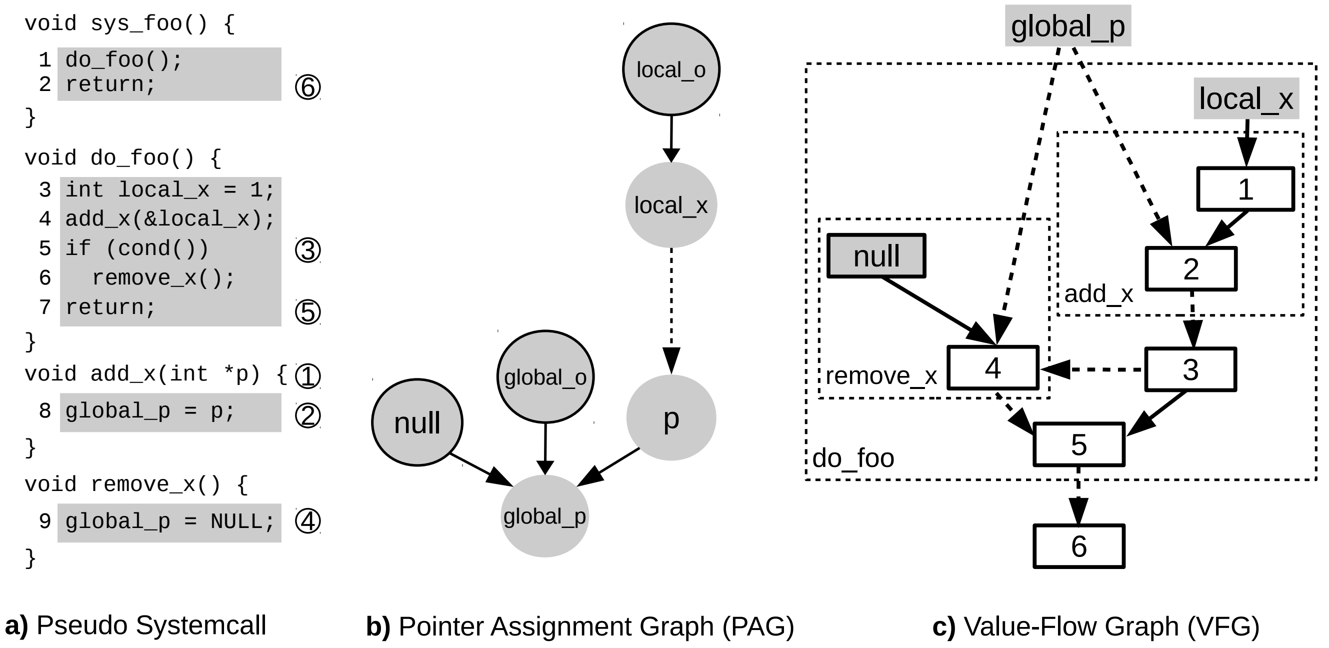
B. Analyzing Kernel Code Per System Call

Although analyzing individual system calls already reduces the amount of relevant code significantly, the resource requirements were still unpractical and we could not collect any data-flow analysis results in our preliminary experiments. For instance, conducting a simple pointer analysis based on this approach already caused our server system to quickly run out of memory (i.e., using more than 32G of RAM). Through careful analysis we found that one of the main causes for the blow-up are function pointers: in particular, the naive approach considers all global variables and functions to be reachable by any system call. While this approximation is certainly safe, it is also inefficient. We use several techniques to improve over this naive approach, which we describe in the following.

Improving Call Graph Accuracy: We start with a simple call-graph analysis, which over-approximates the potential list of target functions. By analyzing the IR of all functions in the call graph we determine if a function pointer is reachable (e.g., by being accessed by a local variable). This allows us to collect possible target functions to improve the precision of the initial call graph. Based on this list, we perform a two-staged pointer analysis in the next step.

Flow-sensitive Pointer-Analysis: To generate the improved call graph we first perform a simple inclusion-based pointer analysis to resolve the constraints of the function pointers collected earlier. To further improve the precision, we conduct a second pointer analysis while also taking the control flow into account. This again minimizes the number of relevant symbols and yields a very accurate context for individual system calls. We store these findings as intermediate results per system call which can be used by subsequent data-flow analysis passes.

Intersecting Global Kernel State: Finally, we combine the previously indentified context information for a system call



```
void sys_foo() {
  do_foo();
  return;             ⑥
}
void do_foo() {
  int local_x = 1;
  add_x(&local_x);
  if (cond())         ③
    remove_x();
  return;             ⑤
}
void add_x(int *p) {  ①
  global_p = p;       ②
}
void remove_x() {
  global_p = NULL;    ④
}
```

a) Pseudo Systemcall **b)** Pointer Assignment Graph (PAG) **c)** Value-Flow Graph (VFG)

Figure 4: Example of a Dangling Pointer vulnerability in a (simplified) system call definition.

with the global kernel context. We do this by determining the global variables of a system call that contain missing references and intersecting these with the list of variables of the global kernel context populated earlier. While possibly increasing the context information our precision improvents prevent an infeasible blow-up in this last step.

C. Minimizing False Positives

False positives are a common problem in static analysis and frequently occur when over-approximating program behavior: for instance, an analysis may assume an alias relationship between pointers that do not co-exist at run time, if the control flow is not taken into account. In the following, we explain how we designed our analysis to be precise and reduce the number of false positives, using dangling pointers as an example. We also provide details on how K-Miner sanitizes the resulting bug reports to further limit the number of false positives.

Precise Data-Flow Analysis: Figure 4 a) shows the code of a pseudo system call with a dangling pointer bug. In step ①, the address of the local variable in do_foo is copied into the parameter p of add_x and subsequently stored in the global pointer global_p in step ②. In step ③, we can see that remove_x will only be called conditionally. Hence, there is a path for which global_p still points to the address of a local variable after execution of do_foo has returned. Looking at the PAG in Figure 4b) reveals that local_o and global_o represent the abstract memory objects behind these possible pointer values. The (simplified) VFG in Figure 4c) shows the corresponding value flows. Our algorithm to find these kinds of bugs consists of two phases: first, we traverse the VFG in forward order starting from local nodes. A reference to a local node leaves its valid scope, if the number of function exits is greater than the number of function entries after traversing the entire path. For the node local_x we can see, that there is one entry to add_x, an exit from add_x, and an exit from do_foo at the end of the path. Consequently, there is a path for which local_x leaves its valid scope, i.e., local_x → ① → ② → ③ → ⑤ → ⑥.

In the second phase we traverse the VFG in backward direction to find (global or local) references to this node, since any such reference represents a dangling pointer. In this case the second phase yields the path ⑥ → ⑤ → ③ → ② → global_p. By querying the PAG dynamically during backward traversal we avoid visiting edges that do not belong to the currently tracked memory location such as ⑤ → ④. This allows us to minimize inaccuracies resulting from over-approximation. We store the respective path information along with the list of nodes and contexts they were visited in as memory-corruption candidate for sanitizing and future reporting.

Sanitizing Potential Reports: Upon completion of the data-flow analysis, we cross-check the resulting candidates for impossible conditions or restrictions which would prevent a path from being taken during run-time execution. Examples for such conditions include impossible call graphs (e.g., call to function g preceding return from function f), or invalid combinations of context and path information. Additionally, we eliminate multiple reports that result in the same nodes for different contexts by combining them into a single report.

D. Efficiently Combining Multiple Analyses

To enable the efficient execution of multiple data-flow analyses, our framework makes heavy use of various optimizations and highly efficient analysis techniques as we describe below.

Using Sparse Analysis: An important data structure in our data-flow analysis is the value-flow graph, which is a di-

				Magnitude of Analysis			Report Results		
Version	MLOC	Bitcode	Avg. Run Time	#Functions	#Variables	#Pointers	DP	UAF	DF
3.19	15.5	280M	796.69s	99K	433K	>5M	7 (40)	3 (131)	1 (13)
4.2	16.3	298M	1435.62s	104K	466K	>6M	11 (46)	2 (106)	0 (19)
4.6	17.1	298M	1502.54s	105K	468K	>6M	3 (50)	2 (104)	0 (31)
4.10	22.1	353M	1312.41s	121K	535K	>7M	1 (30)	2 (105)	0 (22)
4.12	24.1	364M	2164.96s	126K	558K	>7.4M	1 (24)	0 (27)	1 (24)

Table I: Overview of the specifications, resource requirements, and results for the different kernel versions and data-flow passes we used in our evaluation of K-Miner.

rected inter-procedural graph tracking any operations related to pointer variables. The VFG captures the def-use chains of the pointers inside the kernel code to build a *sparse* representation for tracking these accesses. The graph is created in four steps: first, a pointer analysis determines the points-to information of each variable. Second, the indirect definitions and uses of the address-taken variables are determined for certain instructions (e.g. store, load, callsite). These instructions are then annotated with a set of variables that will be either defined or used by this instruction. Third, the functions are transformed into Static Single Assignment form using a standard SSA conversion algorithm [17]. Finally, the VFG is created by connecting the def-use for each SSA variable and made partially context-sensitive by labeling the edges of the callsites. Using this sparse VFG representation in a partially context-sensitive way enables us to conduct precise analysis while reducing the amount of code.

Revisiting Functions: Using different analysis passes, functions are potentially visited multiple times with different values as an input. However, one function might call dozens of other functions and forwarding all the resulting nodes multiple times in the same way would be very inefficient. Our analysis reduces the amount of nodes that have to be forwarded by processing a function only once for all of its possible contexts and storing the intermediate results. If a function entry node is requested by an analysis with a given context, the analysis checks if this node was already visited and re-uses the pre-computed results.

Parallelizing Execution: Because certain analysis steps can actually run independently from each other, we implemented another optimization by parallelizing the forwarding and backwarding processes using OpenMP [4]. OpenMP provides additional compiler directives that allow the definition of parallel regions in the code. In this way, we process some of the heavy container objects used during the analysis in parallel.

V. Evaluation

In this section, we evaluate and test our static analysis framework for real-world operating system kernels. We run our memory-corruption checkers against five different versions of the Linux kernel, using the default configuration (`defconfig`). Our test system running K-Miner features an Intel Xeon E5-4650 with 8 cores clocked at 2.4GHz and 32G of RAM. Table I shows an overview of the analyzed Linux kernel specifications and results: on average, our framework needs around 25 minutes to check a single system call (cf., *Avg. Time* in Table I). This means that a check of the entire system call interface on this server with all three analyses takes between 70 and 200 hours for a single kernel version. [1] In our experiments, K-Miner found 29 possible vulnerabilities, generating 539 alerts in total, most of which were classified as false positives (total alerts are given in parenthesis in Table I). [2] Next, we will evaluate the coverage and impact of those reports and afterwards also discuss the performance of our framework in more detail.

A. Security

Since K-Miner aims to uncover memory-corruption vulnerabilities in the context of system calls, we investigate its security guarantees by inspecting the coverage of the underlying graph structures. To demonstrate practicality, we also present some of the publicly known vulnerabilities we were able to find statically using our framework.

Coverage: Our goal is to uncover all possible sources of memory corruption that are accessible via the system call interface that the kernel exposes to user processes. Hence, we have to ensure that the analysis passes for a certain class of vulnerabilities have access to all relevant information required to safely approximate run-time behavior of the kernel. At the time of writing, our framework contains passes for DP, DF, and UAF, hence, other sources of memory corruption are not covered in this evaluation. However, K-Miner is designed to be extensible and we are working on implementing further analysis passes to cover all remaining vulnerability classes.

The most important factors for the coverage of our three analysis passes are their underlying analysis structures, i.e., PAG, VFG, and context information. Because the inter-procedural value-flow graph and the context information are derived from the pointer-analysis graph, their accuracy directly depends on the construction of the PAG. Our pointer analysis makes two assumptions: 1) partitioning the kernel code along its system call graph is sound, and 2) deriving kernel context

[1] Largely depending on the respective kernel version as seen in the average time per system call in Table I.

[2] Additionally, we are still investigating 158 memory-corruption alerts for the most recent version of Linux.

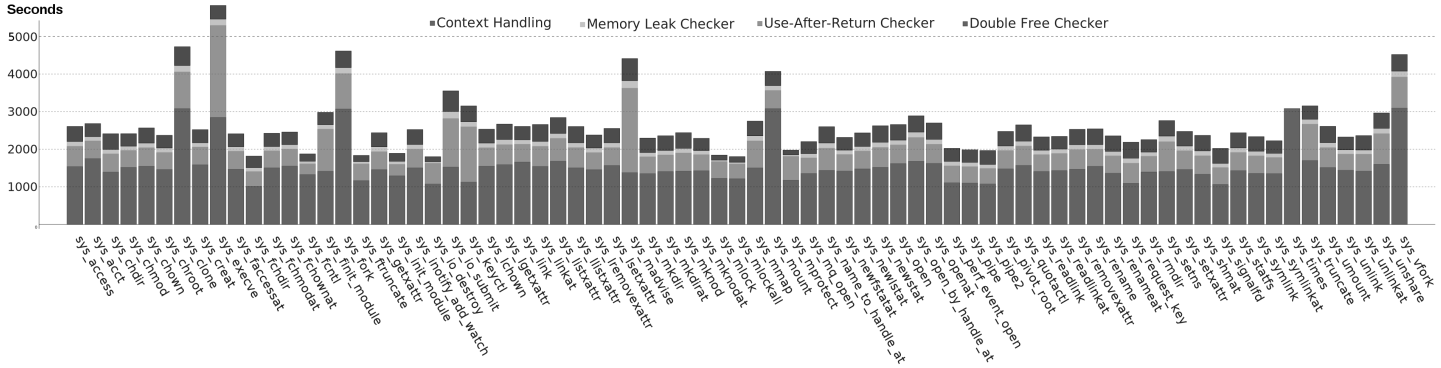


Figure 5: Wall-clock time per analysis phase for system calls requiring more than 30 Minutes within K-Miner.

information from init calls is complete. We would like to note that verifying both assumptions requires a formal proof, which is beyond the scope of this paper. However, we sketch why these assumptions are reasonable in the following.

The first assumption is sensible, because system calls are triggered by individual processes to provide services in a synchronous manner, meaning that the calling process is suspended until execution of the system call finishes. While interrupts and inter-process communication may enable other processes to query the kernel asynchronously, this is orthogonal to the partitioning of kernel code, because these operate under a different context. In particular, our framework allows analyses to take multiple memory contexts into account, e.g., to uncover synchronization errors. Individual analysis passes then have to ensure that the associated contexts are handled accordingly.

Our second assumption is derived from the design of the analyzed kernel code. The init call infrastructure for Linux is quite elaborate, using a hierarchy of different levels that may also specify dependencies on other init calls. Additionally, init calls are used in many different scenarios, e.g., to populate management structures for virtual memory and processes during early boot, or to instantiate drivers and modules at run time. By including all init call levels following their hierarchical ordering in the pre-analysis phase, we ensure that the relevant context information is recorded and initialized accordingly.

Real-world Impact: We cross-checked the reported memory corruptions against publicly available bug reports and found two interesting matches. In particular, our dangling pointer analysis automatically found a bug in Linux kernel version 3.19, which was previously discovered through manual inspection and classified as a security-relevant vulnerability in Linux in 2014 (i.e., CVE-2014-3153). In fact, this vulnerability gained some popularity due to being used as a tool to allow users privilegede access (aka ”root”) on their locked-down Android devices, such as the Samsung S5 [30]. The bug was later discovered to be exploited by the HackingTeam company to spy on freedom fighters and dissidents through a malicious kernel extension [62].

Further, our double-free analysis found a driver bug (i.e., CVE-2015-8962) in a disk protocol driver in version 3.19. The vulnerability allows a local user to escalate privileges and corrupt kernel memory affecting a large range of kernel versions including Android devices such as Google’s PIXEL [26]. Both vulnerabilities were classified as critical issues with a high severity and could have been easily found through K-Miner’s automated analysis. Moreover, we reported two of our use-after-return alerts to the kernel developers.

B. Performance

We now analyze the performance, in particular, the run time, memory consumption, and number of false positives.

Analysis Run Time: As already mentioned, the average analysis time per system call is around 25 minutes. In Figure 5 we give an overview of those system calls for which our analyses took longer than 30 minutes. Most system call analysis are dominated by the context handling. However there are some exceptions, notably `sys_execve`, `sys_madvise`, and `sys_keyctl`. The context handling is time consuming, because it represents the first phase of any subsequent dataflow analysis pass. This means, that it conducts multiple interprocedural pointer analysis, cross-references the global kernel context with the syscall context, and populates the underlying graph data structures for the current system call. This also involves importing and copying information stored on disk, which is slower than accessing RAM. In theory, it should be possible to pre-compute and export the results of the context handling phase for each system call to disk as well. Any dataflow analysis pass could then just re-import the respective file for the current system call, potentially saving some of this overhead (especially in combination with fast SSD hardware). However, we did not implement this optimization feature in the current version of K-Miner.

The UAF checker is notably quicker than the remaining passes, which is due to its reuse of underlying analysis structures from the first pass. In contrast to the use-after-free pass, the double-free analysis has to reconstruct the value-flow graph, which accounts for the majority of its run time. Taken separately, the individual analysis phases require between 5 and 35 minutes run time, which is expected for graph-based analysis, given the magnitude of the input.

Memory Utilization: Initially, our main concern regarded the memory requirements, because of the size of the intermediate representation of the kernel as bitcode image. However, our approach to partition the kernel per system call proved to

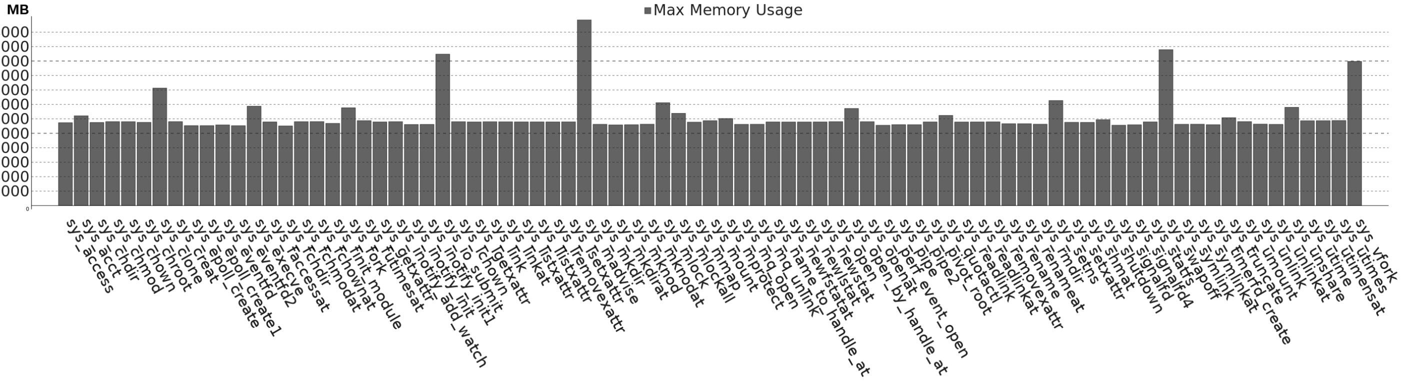


Figure 6: Maximum memory requirements of K-Miner for system calls requiring more than 11G of RAM.

be effective: on average the analyses utilized between 8.7G and 13.2G of RAM, i.e., around a third of our server's memory, with a maximum of 26G (cf., version 4.10 in Table II). We provide a more detailed overview for different system calls in Figure 6. Granted that these numbers also depend to a large extent on the respective kernel version and used configuration, our overall results demonstrate that complex data-flow analysis for OS kernels are feasible and practical. In particular, the memory requirements of K-Miner show that an analysis of future kernel releases is realistic, even with the tendency of newer versions to grow in size.

While the default configuration for the kernel offers a good tradeoff between feature coverage and size, real-world distribution kernels usually have larger configurations, because they enable a majority of features for compatibility reasons. Our current memory utilization is within range of analyzing kernels with such feature models as well. Although we expect to see increased memory requirements (i.e., 128G or more), this does not meet the limit of modern hardware, and K-Miner is able to conduct such analyses without requiring any changes.

C. Usability

Our framework can be integrated into the standard build process for the Linux kernel with some changes to the main build files, which will then generate the required intermediate representation of the kernel image. Using this bitcode image as main input, K-Miner can be configured through a number of command line arguments, such as number of threads, which checkers to use, and output directory for intermediate results. Results are written to a logfile, which can be inspected manually or subsequently rendered using our web interface to get an overview and check reports for false positives.

Version	Avg. Used	Max Used
3.19	8,765.08M	18,073.60M
4.2	13,232.28M	24,466.78M
4.6	11,769.13M	22,929.92M
4.10	12,868.30M	25,187.82M
4.12	13,437.01M	26,404.82M

Table II: Average and maximum memory usage of K-Miner

Reporting Engine: The web interface for our framework is written in Python. It parses the resulting logfile to construct a JSON-based data model for quick graphing and tabular presentation. We attached screenshots in Appendix A to give an impression of an exemplified workflow. While relatively simple, we found this web-based rendering to be extremely helpful in analyzing individual reports. Developers can already classify and comment upon alerts and reports, and we plan to incorporate the possibility to schedule and manage the launch and configuration of analyses from the web interface in future versions.

False Positives: Similar to other static analysis approaches like the Effect-Based Analyzer (EBA) [1], Coccinelle [52], Smatch [9], or APISAN [74], K-Miner naturally exhibits a number of false positives due to the necessary over-approximations. For instance, the use-after-free analysis still shows a high number of false alarms, and leaves room for improvement. In particular, our investigation showed that there are many cases in the kernel code where a conditional branch based on a nullness check is reported as potential use-after-free. Including these cases in our sanitizer component should be straightforward to further reduce this number. However, there will always be a certain number of false positives for any static analysis tool and developers have to cross-check these alerts, similar to how they have to check for compiler-warnings. Overall K-Miner demonstrates that this scenario is practical through some post-processing and intelligent filtering in our web-interface.

VI. Discussion

In this section we discuss our ongoing improvements of K-Miner, various possible extensions, and future work.

A. Soundness

While K-Miner currently does not offer a proof of soundness, we sketched an informal reasoning of why the kernel-code partitioning along the system call API is a sensible strategy in Section V. There are additional challenges for a formal result: first, in some cases the kernel uses non-standard code constructs and custom compiler extensions, which may not be covered by LLVM. However, for these constructs the

LLVM Linux project maintains a list of patches, which have to be applied to the kernel to make it compatible to the LLVM compiler suite. Second, some pointer variables are still handled via `unsigned long` instead of the correct type. These low-level "hacks" are difficult to handle statically, because they exploit knowledge of the address space organization or underlying architecture specifics. Nonetheless, such cases can be handled in principle by embedding the required information in LLVM or by annotating these special cases in the source. Finally, our memory tracking component currently relies on a list of allocation functions. For cases like file descriptors or sockets the respective kernel objects are pre-allocated from globally managed lists and individual objects are retrieved and identified by referring to their ID (usually an integer number). This can be resolved by considering all objects from the same list to be modeled as objects of the same type, and marking functions for retrieval as allocations.

B. Future Work

As K-Miner is designed to be a customizable and extensible framework, implementing additional checkers is straightforward. To this end, we already implemented additional double-lock and memory-leak checkers, thereby covering additional bug classes. Up to this point we only verified that these additional pass implementations are able to detect intra-procedural bugs.[3] However, as our other analysis passes in K-Miner, the double-lock implementation covers inter-procedural double-lock errors in principle, including bugs spanning multiple source files. Similarly, implementing analyses to find buffer overflows, integer overflows, or uninitialized data usage remains as part of our future work to cover all potential sources of memory corruption as mentioned in Section II.

While primarily analyzing the system call API, we found that analyzing the module API in a similar way should be possible and provide interesting results, since many bugs result from (especially out-of-tree) driver and module code. Although this interface is not as strict as the highly standardized system call API, the main top-level functions of many drivers are exported as symbols to the entire kernel image, while internal and helper functions are marked as `static`. Hence, we should be able to automatically detect the main entry points for most major driver modules by looking at its exported symbols and building a call graph that starts with the exported functions. We can then analyze this automatically constructed control flow of the drivers by applying the data-flow analysis passes to the resulting code partitions. In addition to our current approach, this would allow for an extension of our adversary model to include malicious devices and network protocols. We included a prototypical draft of this functionality to analyze module code using K-Miner in the future.

C. Automated Proof-of-Concept Generation

Finding a valid user-space program to provide the necessary input data to reliably trigger a bug is non-trivial in many cases. At the same time, kernel developers will often ignore bug reports without a proof-of-concept. K-Miner's reports already contain all the necessary path information, and hence, it should be feasible to find matching inputs that lead to the execution of that particular path, e.g., by processing the path constraints using a SAT-solver [21]. Alternatively, we could leverage concolic execution [8] or selective, guided fuzzing [58] to generate such proof-of-concepts.

D. Machine Learning

Another possible perspective for interesting future work is to combine our static analysis framework with machine learning, such as deep learning, reinforcement learning, and classifier systems. This would allow for the extraction of common bug patterns and automated pattern mining [73], or scalable classification of generated vulnerability reports, e.g., to build a ranking system for K-Miner's generated reports similar to how APISAN handles the large number of detected semantic function API violations [74]. One of the problems of machine learning approaches is that their results highly depend on the quality of the training data [31].

VII. Related Work

In this section we give a brief overview of the related work and compare K-Miner to existing frameworks and tools. In contrast to dynamic run-time approaches, such as KASAN [39], TypeSan [27], Credal [71], UBSAN [40], and various random testing techniques [34], [35], [22], our approach aims at static analysis of kernel code, i.e., operating solely during compile time. As there already exists a large body of literature around static program analysis [51], [41], we focus on static analysis tools targeting operating system kernels, and data-flow analysis frameworks for user space that influenced the design of K-Miner.

It is important to note that applying static analysis frameworks designed for user space programs is not possible a priori in the kernel setting: data-flow analysis passes expect a top-level function, and an initial program state from which analysis passes can start to propagate value flows. These requirements are naturally satisfied by user space programs by providing a `main` function, and a complete set of defined global variables. However, operating systems are driven by events, such as timer interrupts, exceptions, faults, and traps. Additonally, user space programs can influence kernel execution, e.g., by issuing system calls. Hence, there is no single entry point for data-flow analysis for an operating system. With K-Miner we present the first data-flow analysis framework that is specifically tailored towards this kernel setting.

A. Kernel Static Analysis Frameworks

The Effect-Based Analyzer (EBA) [1] uses a model-checking related, inter-procedural analysis technique to find a pre-compiled list of bug patterns. In particular, it provides a specification language for formulating and finding such patterns. EBA provides lightweight, flow-insensitive analyses, with a focus towards double-lock bugs. Additionally, EBA restricts analysis to individual source files. K-Miner provides an expressive pass infrastucture for implementing many different checkers, and is specifically tailored towards the execution model of the kernel allowing complex, context and flow-sensitive data-flow analyses, potentially spanning the entirety of the kernel image.

[3] In particular, the lock errors introduced in commits `09dc3cf` [53], `e50fb58` [13], `0adb237` [18], and `16da4b1` [2] of Linus' tree.

Coccinelle [52] is an established static analysis tool that is used on a regular basis to analyze and transform series of patches for the kernel. While originally not intended for security analysis, it can be used to conduct text-based pattern matching without the requirement for semantic knowledge or abstract interpretation of the code, resulting in highly efficient and scalable analyses. In comparison to our framework, Coccinelle is not able to conduct any data-flow, or inter-procedural analysis.

The Source-code Matcher (Smatch) [9] is a tool based on Sparse [66], a parser framework for the C language developed exclusively for the Linux kernel. Smatch enriches the Sparse syntax tree with selected semantic information about underlying types and control structures, enabling (limited) data-flow analyses. Like Coccinelle, Smatch is fast, but constrained to intra-procedural checks per source file.

APISAN [74] analyzes function usage patterns in kernel code based on symbolic execution. In contrast to other static analysis approaches, APISAN aims at finding semantic bugs, i.e., program errors resulting from incorrect usage of existing APIs. Because specifying the correct usage patterns manually is not feasible for large code bases, rules are inferred probabilistically, based on the existing usage patterns present in the code (the idea being that correct usage patterns should occur more frequently than incorrect usage patterns). In comparison to K-Miner, APISAN builds on LLVM as well, but only considers the call graph of the kernel and is not able to conduct any inter-procedural data-flow analyses.

TypeChef [38] is an analysis framework targeting large C programs, such as the Linux kernel. In contrast to our work, TypeChef focuses on variability-induced issues and analyzing all possible feature configurations in combination. For this, it provides a variability-aware pre-processor, which extracts the resulting feature model for the kernel, e.g., by treating macros like regular C functions. TypeChef does not conduct any data-flow analysis on their resulting variability-aware syntax tree.

B. User Space Static Analysis

The Clang Static Analyzer [46] consists of a series of checkers that are implemented within the C frontend Clang of the LLVM compiler suite. These checkers are invoked via command-line arguments during program compilation and can easily be extended. As part of the Linux LLVM project [69] there was an effort to implement kernel-specific checkers. However, to the best of our knowledge, this effort has since been abandoned.

The Static Value-Flow (SVF) [59] analysis famework enhances the built-in analysis capabilities of LLVM with an extended pointer analysis and a sparse value-flow graph representation. K-Miner builds on top of LLVM and leverages the pointer analyses provided by SVF to systematically analyze kernel APIs, such as the system call interface.

Mélange [56] is a recent data-flow analysis framework for user space, that is able to conduct complex analyses to find security-sensitive vulnerabilities, such as unitialized reads. Mélange is able to analyze large C and C++ user space code bases such as Chromium, Firefox, and MySQL.

Astrée [15] is a proprietary framework for formal verification of C user programs for embedded systems through elaborate static analysis techniques. It operates on *synchronous programs*, i.e., analyzed code is not allowed to dynamically allocate memory, contain backward branches, union types, or other conflicting side effects. Astrée is able to provably verify the absence of any run-time errors in a program obeying these restrictions and was used to formally verify the primary flight control software of commercial passenger aircraft.

Soot [68] is a popular and widely used static analysis framework capable of conducting extensible and complex data-flow analyses. However, Soot is targeted towards Java programs, and hence cannot analyze programs written in C or C++.

VIII. Conclusion

Memory-corruption vulnerabilities represent an important challenge for the security of today's operating systems. Any instance of one of these bugs exposes the system to a variaty of run-time attacks. Such attacks therefore pose a severe threat to the OS, since they can be launched by unprivileged user processes to exploit a particular vulnerability, e.g., by corrupting memory used by the kernel to gain read and write access to kernel space. This access can then be exploited to escalated privileges of the attacker process to root or achieve arbitrary code execution in the kernel.

Bugs such as dangling pointers, use-after-free, double free, or double-lock errors are introduced through human error during routine programming. Since the code bases of modern kernels are also highly complex, many vulnerabilities are typically hard to find using simple text-based analysis tools. Additionally, finding deeply nested vulnerabilities using dynamic analyses, such as fuzzing, is usually difficult, as every nested layer of call indirection decreases the chances of the fuzzer to randomly trigger the required path. Furthermore, dynamic testing does not offer any guarantees in the case of negative results.

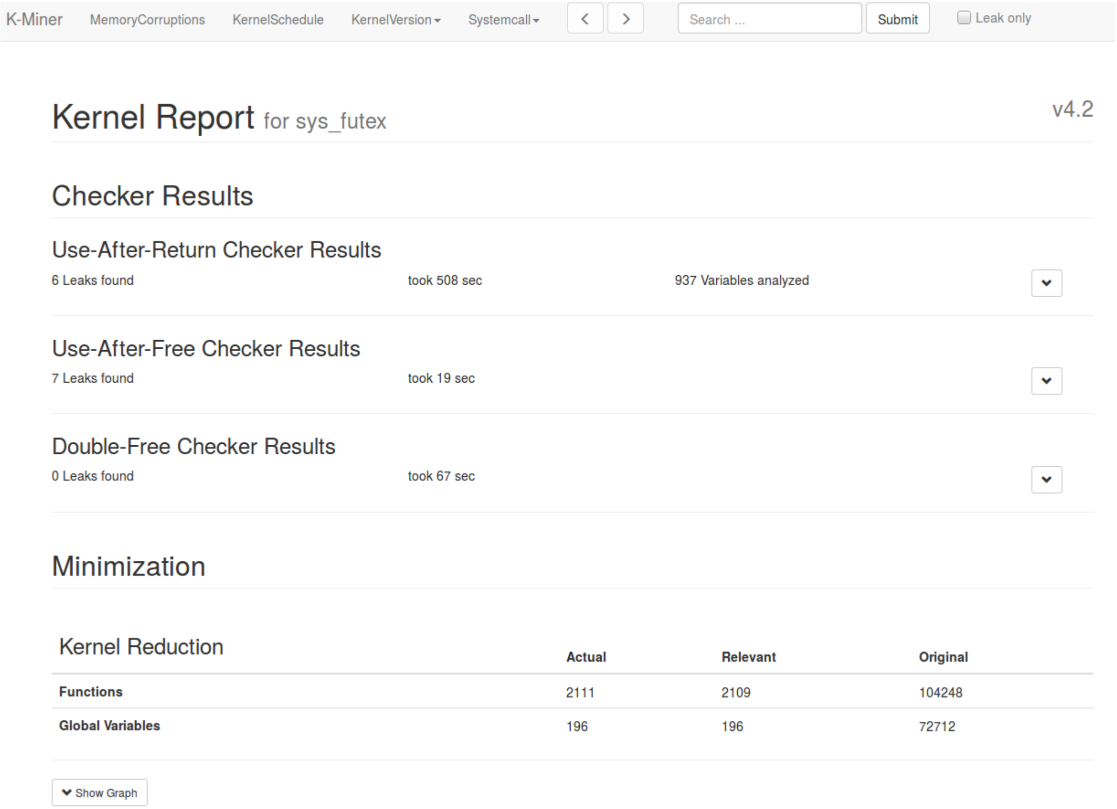
In this paper we present K-Miner, as the first data-flow analysis framework targeting operating systems. We enable scalable, precise, and expressive static analysis for commodity kernels, and demonstrate high practicality by identifying critical vulnerabilities. K-Miner tackles a number of challenges with respect to the huge and highly complex code bases of modern kernels, and provides several analysis passes for finding memory-corruption vulenrabilities such as dangling pointers, use-after-free, double free, and double-lock errors. Our extensive evaluation shows that K-Miner is able to analyze recent versions of Linux in different configurations and uncover real-world vulnerabilities.

Acknowledgement

The authors would like to thank Ferdinand Brasser and Mathias Payer for interesting discussions and comments, as well as the anonymous reviewers for their valuable feedback.

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

Appendix A: Web Interface of K-Miner



Kernel Report for sys_futex

v4.2

Checker Results

Use-After-Return Checker Results

6 Leaks found | took 508 sec | 937 Variables analyzed

Use-After-Free Checker Results

7 Leaks found | took 19 sec

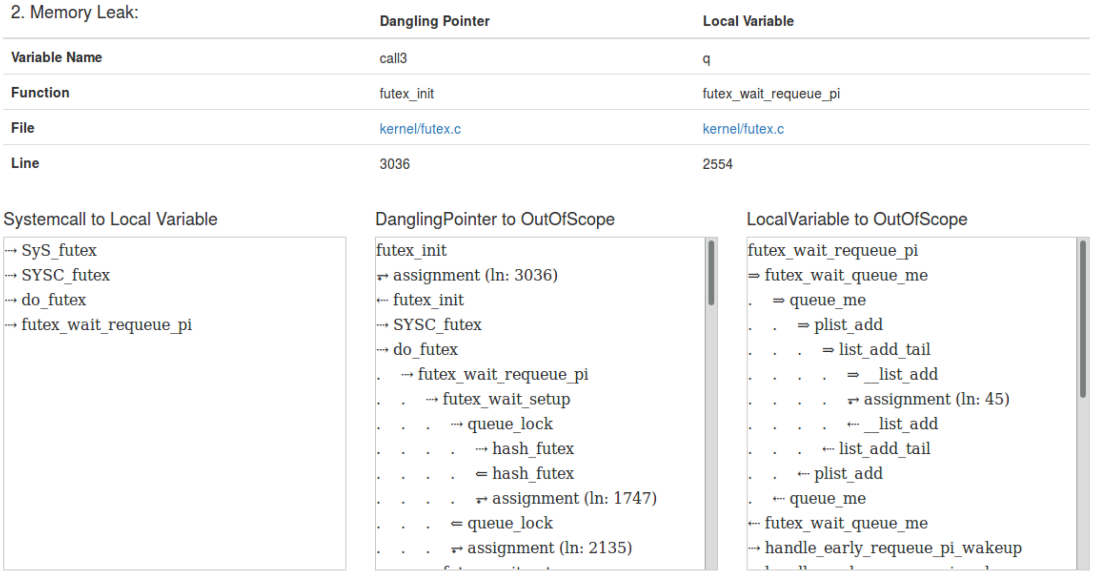
Double-Free Checker Results

0 Leaks found | took 67 sec

Minimization

Kernel Reduction	Actual	Relevant	Original
Functions	2111	2109	104248
Global Variables	196	196	72712

Figure 7: Web interface overview.



2. Memory Leak:	Dangling Pointer	Local Variable
Variable Name	call3	q
Function	futex_init	futex_wait_requeue_pi
File	kernel/futex.c	kernel/futex.c
Line	3036	2554

Systemcall to Local Variable

```
⇢ SyS_futex
⇢ SYSC_futex
⇢ do_futex
⇢ futex_wait_requeue_pi
```

DanglingPointer to OutOfScope

```
futex_init
⥅ assignment (ln: 3036)
⇠ futex_init
⇢ SYSC_futex
⇢ do_futex
.   ⇢ futex_wait_requeue_pi
.   .   ⇢ futex_wait_setup
.   .   .   ⇢ queue_lock
.   .   .   .   ⇢ hash_futex
.   .   .   .   ⇐ hash_futex
.   .   .   .   ⥅ assignment (ln: 1747)
.   .   .   ⇐ queue_lock
.   .   .   ⥅ assignment (ln: 2135)
```

LocalVariable to OutOfScope

```
futex_wait_requeue_pi
⇒ futex_wait_queue_me
.   ⇒ queue_me
.   .   ⇒ plist_add
.   .   .   ⇒ list_add_tail
.   .   .   .   ⇒ __list_add
.   .   .   .   ⥅ assignment (ln: 45)
.   .   .   .   ⇠ __list_add
.   .   .   ⇠ list_add_tail
.   .   ⇠ plist_add
.   ⇠ queue_me
⇠ futex_wait_queue_me
⇢ handle_early_requeue_pi_wakeup
```

Figure 8: Web interface details.

References

[1] I. Abal, C. Brabrand, and A. Wasowski. Effective bug finding in c programs with shape and effect abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 34–54. Springer, 2017.

[2] T. Anton. usb: phy: Fix double lock in otg fsm. https://github.com/torvalds/linux/commit/16da4b1, 2013.

[3] Apple. Kernel extensions. https://developer.apple.com/library/content/documentation/Security/Conceptual/System_Integrity_Protection_Guide/KernelExtensions/KernelExtensions.html, 2014.

[4] ArchitectureReviewBoards. Openmp. http://www.openmp.org/, 2017.

[5] A. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[6] C. Baumann and T. Bormer. Verifying the pikeos microkernel: First results in the verisoft xt avionics project. In *Doctoral Symposium on Systems Software Verification (DS SSV'09) Real Software, Real Problems, Real Solutions*, page 20, 2009.

[7] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Can't touch this: Practical and generic software-only defenses against rowhammer attacks. *arXiv preprint arXiv:1611.08396*, 2016.

[8] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] D. Carpenter. Smatch - the source matcher. http://smatch.sourceforge.net, 2009.

[10] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[11] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, USENIX Sec, 2005.

[12] S. Christey and R. A. Martin. Vulnerability type distributions in cve. *Mitre report, May*, 2007.

[13] H. Christopher. hfsplus: fix double lock typo in ioctl. https://github.com/torvalds/linux/commit/e50fb58, 2010.

[14] J. Corbet and G. Kroah-Hartman. Linux kernel development report 2016. http://go.linuxfoundation.org/linux-kernel-development-report-2016, 2016.

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

[16] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[17] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[18] C. Dan. drm/prime: double lock typo. https://github.com/torvalds/linux/commit/0adb237, 2013.

[19] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. *ACM SIGPLAN Notices*, 50(4):191–206, 2015.

[20] L. Davi, D. Gens, C. Liebchen, and S. Ahmad-Reza. PT-Rand: Practical mitigation of data-only attacks against page tables. In *24th Annual Network and Distributed System Security Symposium*, NDSS, 2017.

[21] L. De Moura and N. Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[22] D. Drysdale. Coverage-guided kernel fuzzing with syzkaller. https://lwn.net/Articles/677764/, 2016.

[23] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *1st IEEE European Symposium on Security and Privacy*, Euro S&P, 2016.

[24] D. Gens, S. Schmitt, L. Davi, and A. Sadeghi. K-Miner source code. https://github.com/ssl-tud/k-miner, 2017.

[25] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium*, USENIX Sec, 2012.

[26] Google, 2016.

[27] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517–528. ACM, 2016.

[28] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 289–298. IEEE, 2011.

[29] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[30] G. Hotz. towelroot by geohot. https://towelroot.com, 2015.

[31] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[32] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

[33] IEEE Computer Society - Austin Joint Working Group. 1003.1-2008 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). http://standards.ieee.org/findstds/standard/1003.1-2008.html, 2008.

[34] D. Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium, pages*, 2011.

[35] M. Jurczyk and G. Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. 2013.

[36] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[37] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security*, pages 957–972, 2014.

[38] A. Kenner, C. Kästner, S. Haase, and T. Leich. Typechef: toward type checking# ifdef variability in c. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 25–32. ACM, 2010.

[39] T. kernel development community. The kernel address sanitizer (kasan). https://www.kernel.org/doc/html/latest/dev-tools/kasan.html, 2014.

[40] T. kernel development community. The undefined behavior sanitizer (ubsan). https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html, 2014.

[41] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[42] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[43] G. Klein. Operating system verification — an overview. *Sādhanā*, 34:27–69, feb 2009.

[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[45] G. Kroah-Hartman. Signed kernel modules. http://www.linuxjournal.com/article/7130, 2004.

[46] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2004.

[47] K. Lu, M.-T. Walter, D. Pfaff, S. Nürnberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. NDSS, 2017.

[48] Microsof. Kernel-mode code signing walkthrough. https://msdn.microsoft.com/en-us/library/windows/hardware/dn653569(v=vs.85).aspx, 2007.

[49] Microsoft. Control flow guard. http://msdn.microsoft.com/en-us/library/Dn919635.aspx, 2015.

[50] MITRE. CVE-2014-3153. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153, 2014.

[51] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[52] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Acm sigops operating systems review*, volume 42, pages 247–260. ACM, 2008.

[53] Z. Peter. printk: avoid double lock acquire. https://github.com/torvalds/linux/commit/09dc3cf, 2011.

[54] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[55] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 2015.

[56] B. Shastry, F. Yamaguchi, K. Rieck, and J.-P. Seifert. Towards vulnerability discovery using staged program analysis. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 78–97. Springer, 2016.

[57] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[58] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[59] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[60] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[61] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[62] H. Team. Hacking team futex exploit. https://wikileaks.org/hackingteam/emails/emailid/312357, 2014.

[63] P. Team. RAP: RIP ROP, 2015.

[64] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 189–200. ACM, 2011.

[65] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *International Conference on Compiler Construction*, pages 17–31. Springer, 2006.

[66] L. Torvalds. Sparse - a semantic parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page, 2006.

[67] W. Trevor. How the linux kernel initcall mechanism works. http://www.compsoc.man.ac.uk/~moz/kernelnewbies/documents/initcall/kernel.html, 2003.

[68] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[69] B. Webster. LLVMLinux. http://llvm.linuxfoundation.org, 2014.

[70] J. Xiao, H. Huang, and H. Wang. Kernel data attack is a realistic security threat. In *International Conference on Security and Privacy in Communication Systems*, pages 135–154. Springer, 2015.

[71] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 529–540. ACM, 2016.

[72] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425. ACM, 2015.

[73] F. Yamaguchi. Pattern-based vulnerability discovery. 2015.

[74] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 363–378. USENIX Association.