École Polytechnique Fédérale de Lausanne

Finding operating system performance knobs using hybrid analysis

by Lucas Lopes Cendes

Master Thesis

Approved by the Examining Committee:

Prof. Dr. Sanidhya Kashyap
Thesis Advisor

Dr. Senbo Fu
External Expert

Vishal Gupta
Thesis Supervisor

Robust Scalable Systems Software Lab
INN 240 (Bâtiment INN)
Station 14
CH-1015 Lausanne

July 7, 2024

Acknowledgments

First, I would like to thank my advisor, Sanidhya Kashyap, for guiding me throughout this project. His competence and expertise in the field were essential for the success of this project. I would also like to thank Vishal Gupta for coming up with the original idea for this project and designing the testing framework used to evaluate this project. His guidance was essential for the development of this project. Furthermore, I would like to thank Yujie Ren for helping me produce the necessary documentation for the tools developed throughout this project. This will ensure that this project can be continued smoothly. Lastly, I would like to thank Mathias Payer for providing the LaTeX template used for this thesis.

Lausanne, July 7, 2024 Lucas Lopes Cendes

Abstract

Finding the optimal configuration parameters in an operating system can significantly improve application performance. Current methods of optimizing such parameters rely on extracting a set of fixed optimal values that cannot be automatically updated at runtime according to the specific workload of an application. A proper framework that can dynamically tune operating system parameters at runtime would first need to devise a way of detecting accesses to these parameters. In this thesis, we designed a system that can detect references to sysctl knobs in the Linux kernel by using the information extracted through the static analysis of the kernel's source code. Our static analyzer locates the variable used to store the value of a particular knob, propagates its value across variable assignments and function calls, and builds a call graph of functions that reference the value of the knob in question. Our dynamic detection tool then uses these call graphs to generate an eBPF program that detects the respective function calls by hooking the relevant functions in the call graph. Out of a total of 241 sysctl knobs tested, we were able to locate 201 knobs in the Linux kernel and generate call graphs for 118 of those. We tested our dynamic detection routing with Memcached and Redis and were able to detect references to 35 knobs for Memcached and one knob for Redis.

Contents

Acknowledgments **1**

Abstract **2**

1 Introduction **5**

2 Background **8**
2.1 Sysctl 8
2.2 Cscope 9
2.3 eBPF 10
2.4 Kallsyms 10
2.5 The __init attribute 11

3 Design **12**
3.1 Static Analysis 12
3.1.1 Data Structures 13
3.1.1.1 Struct Hierarchy 13
3.1.1.2 Call Graph 14
3.1.2 Value Propagation 14
3.1.2.1 Variable Assignment 15
3.1.2.2 Function Calls 16
3.1.2.3 Function Returns 19
3.1.3 Call graph expansion 20
3.2 Dynamic Detection 21

4 Implementation **23**
4.1 Extracting Complete Expressions 23
4.2 Finding Sysctl Knob Variables 25
4.3 Filtering References to Variables 26
4.3.1 Detecting Variable Declarations 26
4.3.2 Determening The Locality of Query Results 28

4.4 Finding Function Declarations 28
4.5 Database 29
4.5.1 Function-Value Query Table 29
4.5.2 Function Declaration Table 29
4.5.3 Macro Return Ranges Table 30

5 Evaluation **31**
5.1 Results 31
5.1.1 Static Analysis 31
5.1.2 Dynamic Detection 32
5.2 Discussion 33

6 Conclusion **36**

Bibliography **37**

Chapter 1

Introduction

The operating system is often a considerable bottleneck in the performance of applications. Misconfigurations may result in a significant loss of performance of some key operations of an operating system [15]. As such, deriving optimal configurations is essential for ensuring that applications run efficiently.

The number of configurable parameters in an operating system is often extremely large, so deriving the optimal configuration can be challenging [18] [9]. In addition, determining the set of relevant parameters for a specific application is often difficult as this may depend on information only available at runtime and may change according to the workload [20]. Furthermore, different applications have different requirements, meaning a global set of configurations may not achieve optimal performance. Several approaches have been taken to derive the optimal settings for an operating system.

Cocoon attempts to tailor an operating system to a specific architecture by compiling it on boot time. It achieves this by collecting information about the hardware of the target system and using it to apply architecture-specific optimizations during the compilation phase. This process requires either the source code or an intermediate representation (IR) of the operating system to be stored in the boot partition. Cocoon was tested for both the Linux kernel and FreeBSD, resulting in better or equal performance than the standard compiled image of the operating system in question [7]. However, this approach does not consider the performance of specific applications or information available during runtime.

Wayfind attempts to determine optimal operating system parameter configurations by testing different values and using a given application's performance metrics as feedback. It takes a set of configurations to be tested as an input and uses different machine-learning methods to explore different values for the given parameters until the optimal configuration is determined. To accomplish this, it runs a series of test jobs for each set of parameter values, collects the respective performance metrics, and uses them as feedback in its machine-learning model to determine the next set of values to be tested. Their approach, however, still requires the user to specify the list of parameters to be tested, and the set of parameter values is fixed after Wayfind finishes running its set of tests [9]. Because of that, it cannot dynamically modify the set of parameters according to the specific workload of an application at a given time.

The previously mentioned works derive a static configuration that is not modified after their evaluation steps. As a result, information related to an application's specific runtime conditions, which may change with different workloads, is not considered. As such, optimization opportunities that adapt to these conditions are missed. This can be accounted for by performing continuous optimizations on a running process [20].

OCOLOS is the first attempt at an online optimization system that continuously modifies a running application according to performance metrics collected at runtime. It can take an unmodified standard binary and modify its code at runtime. It does this by continuously performing a series of performance measures and analyzing the access patterns of a running program. The optimizer then uses this information to produce an optimized version of the application's machine code and injects it into the code portion of the address space of the running process. References to the old un-optimized code are then redirected to point to the new code produced by OCOLOS. By doing this, it can continuously adapt an application to the specific runtime conditions that may change over time and vary according to the specific workload of the application [20]. This approach, however, only considers the application itself and does not account for bottlenecks introduced by the operating system.

The online optimization of operating system parameters can complement systems like OCOLOS to achieve optimal application performance. To accomplish this, we must first devise a way of detecting when a given operating system parameter is referenced while servicing the system calls executed by an application. This process would allow us to determine which parameters affect a running application's performance. In Linux, for example, we must be able to detect references to sysctl knobs at run time.

Currently, it is possible to detect when a sysctl knob in Linux is read or modified by a userspace application. This detection can be done using eBPF to hook the sysctl subsystem in the Linux kernel [2]. There is, however, no way to detect when the kernel itself accesses a given sysctl knob, nor is there a way to determine all the memory locations in which the value of a sysctl knob is stored.

In this thesis, we describe a system that can detect references to sysctl knobs in the Linux kernel by analyzing call graphs produced through the static analysis of the kernel's source code. In particular, the static analyzer is able to locate the references to a given sysctl knob, propagate its value across variable assignments and function calls and generate a call graph of the functions that access the value of a particular knob. This call graph is then used to produce an eBPF program that detects when a given call chain in a call graph is executed. With that, we can detect the possible usage of a sysctl knob.

The static analysis step of our system takes the source code of the Linux kernel as an input, finds the variable in which a sysctl knob is stored, and determines the other variables to which the value of the knob is propagated. It does this by parsing the set of assignments, function calls, and return statements that involve the variable in question. It then uses this information to detect references to the value of a knob and builds a call graph of the functions that must be traversed to access the given value. This graph is then saved into a file that can be loaded by our dynamic detection system.

The dynamic detection step uses the call graph produced by the static analysis to generate an eBPF program that hooks the relevant functions in the call graph. In particular, the program hooks the entry point and the last function in a given call chain. When an entry point function is executed, a flag is set. When the last function in a call chain is executed, the flag of one of its entry points is checked, and if it is set, we count this as access to the corresponding knob. The entry point flag is then cleared when the entry point function returns.

Out of the 241 sysctl knobs we chose for our analysis, our program was able to locate the respective variable for 201 of them and generate call graphs for 118 of them. Our dynamic detection routine was able to detect references to 39 different knobs for Memcached and one knob for Redis. Most of the knobs detected for those applications were related to virtual memory, scheduling, and network device management.

Our key contributions to this project are a static analyzer tool that can parse the Linux kernel's source code and a tool that generates eBPF programs that hook different functions. We describe the concepts explored in this thesis in chapter 2. We then explain the architecture of our static analysis and dynamic detection routines in chapter 3. The specific implementation details of the static analyzer are then described in chapter 4. The results we obtained are presented, and the limitations of our system are described in chapter 5. We then conclude this thesis in chapter 6.

Chapter 2

Background

In this chapter, we will introduce some of the key concepts explored throughout the thesis. Section 2.1 explains the Sysctl mechanism, which is the main focus of this thesis. Section 2.2 introduces the cscope tool, which is used extensively in the static analysis procedure described in section 3.1. Section 2.3 explains the eBPF technology used in the dynamic detection procedure described in the 3.2. Sections 2.4 and 2.5 explain the kallsyms mechanism and the `__init` attribute in the Linux kernel, respectively, which are both relevant to the dynamic analysis procedure in 2.5.

2.1 Sysctl

The sysctl mechanism in the Linux kernel exposes certain kernel parameters to user mode, allowing users to read and modify them at runtime. These parameters are known as knobs and are used across the different subsystems in the Linux kernel. [17]. Each knob is given a hierarchical name corresponding to the path of the source code of the subsystem in which it is used. For example, the knob `net.core.busy_poll` is used in the subsystem defined in the net/core directory of the source code of the Linux Kernel [3]. Sysctl knobs are stored in special structures in the Linux kernel.

The sysctl knobs are identified through entries in structures known as `ctl_tables`. An example of an entry in a `ctl_tables` is shown in figure 2.1. The `.procname` field contains the leaf name of a sysctl knob. For example, the `ctl_table` entry for the knob `net.core.busy_poll` has its `.procname` field set to the string literal `"busy_poll"`. The `.data` field of the entry may contain the name of the variable in which this sysctl knob is stored. This field may be omitted or set to NULL. The `.proc_handler` is set to the name of the handler function that is used to read and set the value of the respective knob. This function might be one of the generic handler functions defined in `kernel/sysctl.c` or might be a custom handler that is specific to a given knob. Each subsystem stores its respective `ctl_table` in structures known as sysctl tables, which are arrays of `ctl_table` entries [19].

```
        .procname       = "busy_poll",
        .data           = &sysctl_net_busy_poll,
        .maxlen         = sizeof(unsigned int),
        .mode           = 0644,
        .proc_handler   = proc_dointvec_minmax,
        .extra1         = SYSCTL_ZERO,
```

Figure 2.1: The `ctl_table` definition for the `net.core.busy_poll` sysctl knob [16].

2.2 Cscope

Cscope is a tool for searching for symbols or strings in C source code files. It uses a symbol cross-reference table containing the locations of the different variables, functions, function calls, and macros within the source code files being searched. This cross-reference table is built the first time a given set of source code files is searched and updated whenever the respective source code files are changed. The header files in the default include directory are also added to the cross-reference table unless a kernel mode flag is specified [1].

Cscope supports different types of search queries that handle the different types of symbols present in C source code files. Three types of queries are relevant for our purposes. The first one is searching for a C symbol, which allows us to search for a variable, function, or macro with the specified name. Search results will include both the declarations and references of the specified symbol. The second query type allows us to search for the declarations of global symbols, such as functions and global variables. The third type of query enables us to search for occurrences of a given string in a set of source code files, which allows us to search for literals. Query results refer to the entire source code tree being searched [1].

There is no way of limiting the scope of a cscope search within a given source code tree without modifying the structure of the source code tree itself. As such, all symbols with the same name declared in different scopes will be shown in the query results for a given symbol name. In addition, it is impossible to search for declarations of local variables specifically. Because of that, it is up to the user to determine the query results that refer to a specific scope.

The query results of a cscope search follow a specific format that can be divided into four space-separated segments. Figure 2.2 shows an example of this. The first segment is the source code file path to which a query result refers. The second segment is the name of the function in which the given reference is found. References that refer to the global scope and are not contained inside functions have their function field set to `<global>`. The third segment is the line number within the respective source code file to which a query result refers. The fourth and last segment is the line of code itself. Query results are separated in different lines [1].

```
fs/lockd/svc.c lockd 139 struct net *net = &init_net;
```

Figure 2.2: Example of a cscope query result for the symbol `init_net` in the Linux kernel source code.

2.3 eBPF

The eBPF technology allows sandboxed programs to run within the Linux kernel. Thus, it effectively allows us to extend the kernel's functionality without modifying it. eBPF programs are initially compiled into bytecode and then translated into machine code by a just-in-time (JIT) compiler. Programs must pass through a verification step before they are loaded into the kernel to ensure that the stability of the kernel is not compromised. After being loaded, an eBPF program is hooked to a specific code path in the kernel and is executed whenever the given code path is traversed [6] [5].

eBPF programs may be attached to a set of different hook points. The two relevant types of hook points for our purposes are `kprobes` and `kretprobes`. A `kprobe` hook point allows a program to be hooked to a specific instruction within the kernel. The eBPF program is then run whenever the instruction at the specified address is executed. One of the most common uses of `kprobe` hook points is to hook specific functions in the kernel such that an eBPF program is executed whenever a given kernel function is entered. A `kretprobe` allows an eBPF program to be hooked to a function's return, meaning that the program will execute whenever the function returns. eBPF programs are event-driven and use special data structures to store state [6] [10].

One of the main data structures used in eBPF programs is the eBPF map. This structure allows data to be shared between different eBPF programs and with applications running in userspace. These maps can be accessed through a set of helper functions defined in the eBPF library. Different types of maps are supported [5]. One of the simplest types of maps is the array map, which provides generic array storage and allows elements to be accessed through an index. An array map may also be defined as a per-CPU map in which different slots are allocated to different CPUs. A program accessing a per-CPU map through a helper function will always access the slot in the corresponding CPU in which it is running [8]. Maps have a fixed size that is set when they are declared [5].

2.4 Kallsyms

The kallsyms mechanism exposes all global symbols in the Linux kernel. It can, in particular, expose the names of all non-inlined kernel functions. This mechanism can only be used if the kernel is compiled with custom flags, as the symbols are not included in the kernel image by default. A list of all the symbols and their respective addresses can be accessed through the `/proc/kallsyms` file. This allows us to resolve the address of a given kernel function, which is especially useful for attaching krpobes to specific functions [13] [10].

2.5 The __init attribute

The `__init` attribute in the Linux kernel code is associated with functions that are a part of its initialization process. A function marked with this attribute is discarded and has its memory freed once after it returns [14].

Chapter 3

Design

The workflow of our design can be divided into two parts, namely, the static analysis of the source code of the Linux kernel and the detection of the usage of sysctl knobs at runtime. The static analysis identifies all of the references to the value of a given knob and produces a call graph of the set of functions that access the value. The dynamic detection of the usage of the knobs uses the call graph produced in the static analysis to produce a simple eBPF program that hooks the functions that reference the knob. The static analysis step is the main focus of this project and will be described in detail in the section 3.1. The dynamic detection step will be explained in section 3.2

3.1 Static Analysis

We start the static analysis by determining the variable in which the value of a given sysctl knob is stored. We then look up all of the references to this variable in the source code of the Linux kernel. Since knobs are often stored in structs, we use the **struct hierarchy** of a given variable to determine if the value of a knob is referenced. This data structure contains the set of struct fields that need to be referenced to retrieve the value of a knob from a given struct. If a function references the complete struct hierarchy of a variable, we count that as a match and add the given function as a root in our call graph. Since the value of a knob can be propagated through a set of variable assignments and function calls, we must take those into account when performing a knob search.

We can handle the propagation of a given knob value through variable assignments and function calls by searching for the respective assigned variable or function argument. When handling variable assignments, we must determine if the assigned variable is local or global. We can restrict our search to one specific function for local variables, but we have to keep track of assignments to the return arguments of the given function. For global variables, we must consider the return values and return arguments of all functions that reference the variable. When handling function calls, we must account for the struct hierarchy of the return values, as well as the set of relevant return arguments and their respective struct hierarchies. We also need to handle the propagation of values through nested function calls and keep track of the function pointers passed as arguments to function calls. Furthermore, if we identify a match to a sysctl knob in a given function, we must propagate that through the call chain and add the respective callers to our call graph. With this process, we can build an initial call graph representing the set of function calls that are needed to access the value of a particular knob.

When performing a global variable search, we must also be able to propagate the value of a given knob through the functions at the top of the call graph, which we will describe as **entry points**. This propagation is achieved by keeping track of relevant return values and return arguments of the function that reference the respective global variable. With that, we can propagate the respective return values and output parameters through the callers of the entry points and expand our set of entry points. We can repeat this process recursively to propagate a value up the call chain. The resulting call graph can be expanded even further by considering the callers of the expanded set of entry-point functions.

The final step of our variable search process is identifying the complete set of callers of the endpoint functions. This process is repeated recursively until either we reach the top of the call chain or a specified depth limit is reached. Our final call graph is then saved to a file that can later be used to perform the dynamic detection of the usage of sysctl knobs.

In the next subsection, we will describe the different steps of our static analysis in more detail. We will start by describing our basic data structures, such as the struct hierarchy and call graph, in section 3.1.1. We will then explain how we handle the propagation of values through assignments, function calls, and function returns in section 3.1.2. Finally, section 3.1.3 will explain how we expand the initial call graph build in the value propagation phase.

3.1.1 Data Structures

Two data structure types, the struct hierarchy, and the call graph, are used throughout the static analysis phase. When querying a variable, the struct hierarchy of each query result determines whether the given reference should be considered. The call graph is built throughout the static analysis phase and is the main output of this phase.

3.1.1.1 Struct Hierarchy

The struct hierarchy is an ordered list of the struct fields of the struct variable accessed in a given reference. No distinction is made between direct access to a field through the dot (.) operator or dereferencing a pointer stored in a field with the arrow (->) operator. Consider the example shown in figure 3.1. If we have a reference to `a->b.c->d`, we would count that as a reference to variable `a` with a struct hierarchy represented by the list `{b, c, d}`. References to a variable `x` that is not a struct would be classified as having an empty list as their struct hierarchy. This approach allows us to extract the name of the referenced struct variable and filter its query results according to the respective struct hierarchies being referenced.

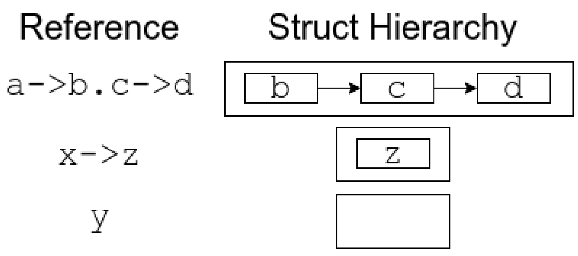

Figure 3.1: Examples of struct hierarchies for different references.

When searching for a variable with a given struct hierarchy, we define the **struct match** of a given query result as the index right after the last field in the desired struct hierarchy accessed in a reference. If a reference accesses at least one struct field not present in the desired struct hierarchy, we say this reference has no struct match and can be discarded. For example, if we are searching for variable `a` with a struct hierarchy of `{b, c, d}` and we see a reference to `a->b`, we say that this has a struct match of 1. Likewise, if we see a reference to `a` with no struct fields, we consider a struct match of 0. If there is, however, a reference to `a->b->e`, we say that this reference has no struct match since it accesses a struct field `e` that is not present in the desired struct hierarchy. We will say that a reference has a **full struct match** if its struct match is equal to the length of the struct hierarchy, meaning that all struct fields in the hierarchy are accessed. With this strategy in place, we can filter out references that are irrelevant to our search and determine if a reference is accessing the value being searched.

3.1.1.2 Call Graph

The call graph that is built in the static analysis step represents the chain of function calls that are needed to access the value of a given knob. The nodes in the call graph represent the functions, with directed edges going from the callee to a caller. The roots of the call graph are the functions that contain at least one reference with a full struct match of the struct hierarchy of the variable currently being searched, meaning that they directly access the desired variable. An entry-point function is the first function in its call chain that has at least one reference with a struct match. All the callers of the entry point functions are in the call graph. These functions have no struct matches with the desired struct hierarchy and are added to the graph in the graph expansion phase. A function can be both an entry point and a root if it is the only function in its call chain that contains a reference with a struct match. Figure 3.2 shows an example of a call graph. Every call chain contains a root and an entry point.

3.1.2 Value Propagation

The value of a sysctl knob may be propagated across a series of function calls and variable assignments. After submitting a search query for a given variable, we check if the given reference contains

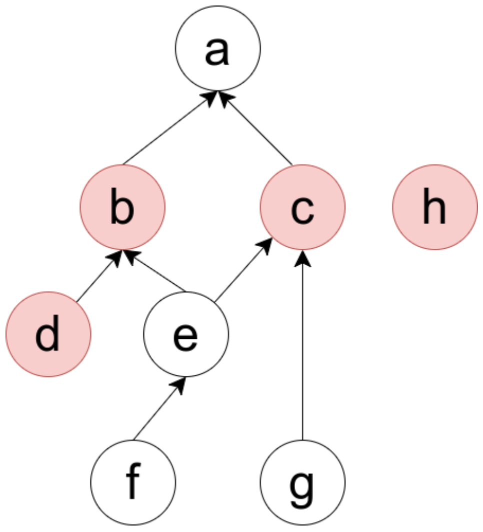

Figure 3.2: Example of a call graph. Functions d, f, g, and h are roots, and functions b, c, d, and h, colored in red, are entry points. Functions d and h are both entry points and roots

a function call, and if that is the case, we call our function call handling routine. Otherwise, we check if the reference contains an assignment of the variable being queried to another variable. If so, we update the struct hierarchy accordingly and search the assigned variable. After executing either of those routines, we check if the current reference is a return statement that contains the desired variable. If that is the case, we check whether the statement returns the value of a variable or the return value of a function call. If it is a value return, we use the struct hierarchy of the value in the return statement to determine the **return struct hierarchy** of the current function. Otherwise, if the return value of a function call is returned, we use the return struct hierarchy obtained by our function call routine as our return struct hierarchy. The next sections will describe each one of those steps in more detail.

3.1.2.1 Variable Assignment

We consider an assignment statement to be an assignment of the current value being searched to a different variable if the right-hand side of the current statement has a struct match with the struct hierarchy currently being searched. If that is the case, we must extract the assigned variable's name and the assigned variable's desired struct hierarchy. Figure 3.3 shows an example of how the new struct hierarchy is determined. Consider the case where we are searching for variable `a` with a struct hierarchy of `{b, c, d}`, and we encounter the assignment statement `x->y = a->b.c`. To determine the assigned struct hierarchy, we first get the struct hierarchy of the left-hand size, which in this case is `{y}`. We then get the struct match of the right-hand side and remove all segments of the struct hierarchy that come before the index represented by the struct match. In our example, since our current struct hierarchy is `{b, c, d}` and the struct match of the right-hand side of the expression is 2, the resulting struct hierarchy is `{d}`. Finally, we append the right-hand size struct hierarchy to the one obtained on the left-hand side of the expression, meaning that the assigned variable is `x` with a struct hierarchy of `{y, d}`. We then proceed by starting a new search for this assigned variable.

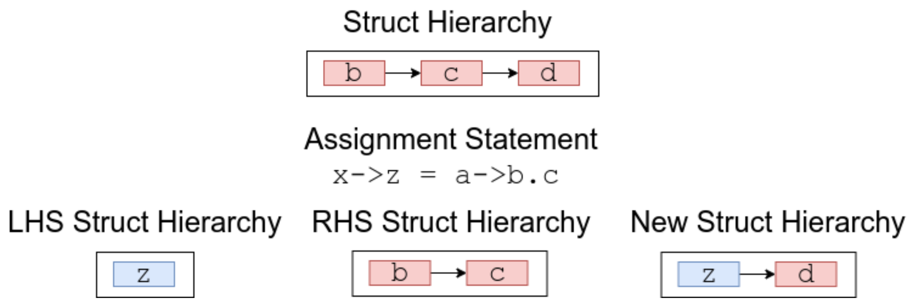

Figure 3.3: Example of how an assignment is handled. In this case, we are searching for the variable `a` with the struct hierarchy shown above. After encountering the assignment statement shown in the middle, we would start a search for the variable `x` with the new struct hierarchy shown at the bottom.

When handling variable assignments, we must also check if it is assigned to a return parameter of the current function. As such, we must first check if the left-hand side of the expression contains a dereferenced pointer, either through the start (*) operator or a dereferenced struct field accessed with the arrow (->) operator. If that is the case, we must check if the respective variable is listed as a parameter in the function declaration of the current function. If so, we append the assigned variable and the respective struct hierarchy to the return parameter list of the current function.

After determining the assigned variable, we must determine if the respective variable is either global or local. This is done by checking if a variable declaration exists within the current function, either as a part of the parameter declarations in the function's declaration or as a variable declaration inside the function's body. If that is the case, we consider this variable to be local and only consider the references that are a part of the current function. Otherwise, if no function declaration is found inside the current function, the variable is considered global, and all references obtained for the current variable are considered. Determining whether an assigned variable is global or local is also used to determine how the return struct hierarchies and output argument lists obtained in the search for the assigned variable are treated, as it will be explained in section 3.1.2.3.

3.1.2.2 Function Calls

If an expression contains at least one function call, we must first obtain the names of all the functions in the current expression and determine which arguments contain the value being searched. The names of the functions are added to a list in the reverse order in which they appear in the current expression to ensure that nested function calls are handled first. Then, for each function, we obtain the struct matches of each argument. If an argument has a struct match, the index of the respective argument is added to the list of function arguments that will be searched. The struct hierarchy of each searched argument is determined in the same manner described in section 3.1.2.1. Since there are no assigned struct fields to consider, we set the new struct hierarchy to the list of struct segments with an index greater than or equal to the struct match of the given argument. Figure 3.4 shows an example of this. After the list of function calls and the respective arguments are obtained, we resolve each function to proceed with the search.

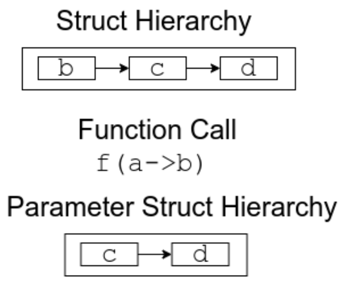

Figure 3.4: Example of how the struct hierarchy of a function parameter is computed. In this case, we are searching for the variable `a` with the struct hierarchy shown above. When we encounter the function call shown in the middle, we would start a search for the respective parameter in function `f` with the struct hierarchy shown below.

After obtaining the name of the function being called, we must find the declaration of the respective function and extract the names of the function's parameters. This process is done by performing a cscope query for the respective function name and determining which of the results is the declaration of the function, with header declarations being discarded. Since multiple functions can have the same name, we must determine which declaration is in scope to identify the actual function being called. When this process is done, we match the indices of the arguments that will be searched with the respective parameter names and determine the indices of the parameters that correspond to function pointers.

After obtaining the function declaration, we must be able to resolve the names of the functions being passed as function pointers. To do that, we create a function pointer hash map that maps the name of the function pointer parameter to an entry containing the name of the respective parameter and the path to the calling function's source file. To do this, we first check if the function pointer map passed by the caller contains an entry for the argument being passed to the callee. If that is the case, we retrieve the entry and create a new mapping between the respective name of the parameter of the callee and the entry. Otherwise, we create a new entry containing the name of the argument being passed and the source file path of the current function and map it to the name of the callee's parameter. This procedure is illustrated in figure 3.5. When handling function calls for the callee, we will first check if the function call name corresponds to an entry in the function pointer map, and if that is the case, we will use the name stored in the map entry as the name of the function being called. We will also use the source file path stored in the entry instead of the source file path of the current function to determine if a function declaration is in scope. We then will proceed with the search for the function call parameters.

After matching the parameter names with the respective arguments containing the values being

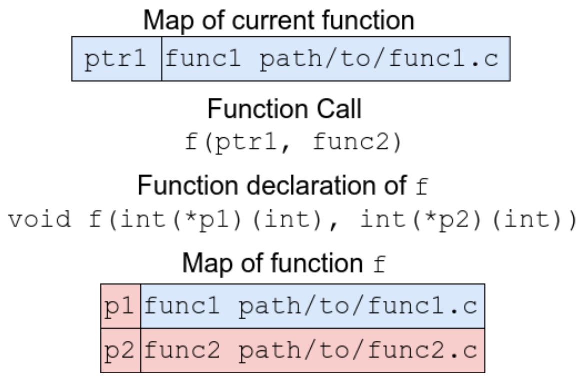

Figure 3.5: Example of how a function pointer map is produced. In this case, we are evaluating the current function, and we encounter a function call to function `f`.

searched, we must perform a query for the parameters in question. This search is always considered a local variable search, meaning we can remove query results that are not in the called function. We then proceed by parsing the references for each parameter and respective struct hierarchy. After that, we obtain the return struct hierarchy and the list of relevant return parameters for the function being called. Afterward, we check if there is a full struct match in one of the functions in the call chain obtained when evaluating the function call. If that is the case, we add a node for the current function to the call graph and insert an edge going from the function that was called to the current function. We then continue by handling the list of return parameters.

We start evaluating the list of return parameters obtained after evaluating a function call by matching the names of the parameters to the respective arguments passed to the function. For each return parameter, we obtain the struct hierarchy of the passed argument and the struct hierarchy assigned to the respective parameter. Since this can be treated as an assignment of the parameter value to the output argument passed, we can generate the assigned struct hierarchy and search the respective return argument in the same way described in section 3.1.2.1. A special case must be considered when a nested function is passed as the output argument. In this case, we must search for each parameter in the nested function, using an empty struct hierarchy for each. We then determine which one of the arguments contains the value of the pointer returned by the nested function. We then use the respective argument name and the struct hierarchy returned by the nested function as the output argument's respective variables and struct hierarchies. From there, we can proceed like in section 3.1.2.1. Afterward, we must check if the output argument is a parameter of the current function. If that is the case, we must append the output argument and the respective assigned hierarchy to the output parameter list of the current function. We then handle the return struct hierarchy obtained for the called function.

If a valid return struct hierarchy was obtained for the function that was called, meaning that the return statement of the callee had a struct match, we must then handle the different ways this return value can be used. We first check if the function call is on the right-hand side of an assignment, and if that is the case, we generate the resulting assigned struct hierarchy and search the assigned variable in the same way described in the section 3.1.2.1. Afterward, we check if the function call is nested as an argument to another function call. In this case, we must find the index of the respective function in the function call list that was produced earlier and retrieve the respective argument list. Afterward, we append the index of the argument corresponding to the nested function call to the list of arguments that will be searched and set the struct hierarchy of the corresponding argument to the return struct hierarchy of the nested function call. Since nested function calls are always evaluated first, the respective return hierarchy of the nested call can be evaluated appropriately. Otherwise, if the return struct hierarchy corresponds to the outermost function in the call, we use it as the overall function call return hierarchy. This return hierarchy will be potentially used as the return struct hierarchy of the current function if the function call is a part of the return statement of the current function, as it will be described in 3.1.2.3. The procedure described in the last few lines applies to almost all function calls, with the only notable exception being `memcpy`, which must be handled differently.

Function calls to `memcpy` are a special case as they can be considered to be an assignment from the `src` argument to the `dest` argument. We only evaluate function calls to `memcpy` if the `src` argument has a struct match with the current struct hierarchy. If that is the case, we proceed in the exact same manner as described in section 3.1.2.1. If a nested function call is passed as the argument to the `dest` parameter, we can obtain the name of the pointer returned and the respective struct hierarchy in the same way as it is done for output arguments of regular function calls.

3.1.2.3 Function Returns

If a reference is a return statement that references the current value being searched, we must obtain the struct hierarchy of the returned value. We will refer to this hierarchy as the **return struct hierarchy** of the current function. If the return statement contains at least one reference to the searched value, we obtain the return hierarchy in the same way as it is done for function arguments, as described in section 3.1.2.2. Figure 3.6 shows an example of this. If the return statement only contains a function call, we use the return hierarchy obtained in the function call handling routine described in 3.1.2.2 as the return hierarchy of the current function. The return struct hierarchy is propagated differently depending on whether the current variable being searched is local or global.

When searching local variables, only a single struct hierarchy for the current function is propagated across searches for different variables within the same function. For variable assignment, the valid return struct hierarchy obtained after searching the assigned variable is used as the return struct hierarchy of the current function. For function calls, the return struct hierarchy obtained when handling output arguments or the one obtained for a variable assignment to the return value of a function call is used as the return struct hierarchy of the current function. The return struct hierarchy of the function call itself may be propagated if the function call is in a return statement, as discussed earlier. If multiple valid return struct hierarchies exist for the same function, the one

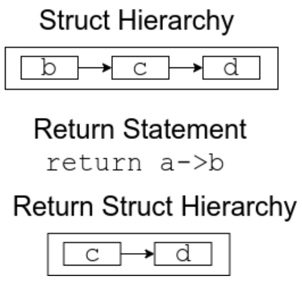

Figure 3.6: Example of how a return struct hierarchy is computed. In this case, we are searching for variable a with the struct hierarchy shown above. When we encounter the return statement shown in the middle, we would set the return struct hierarchy of the current function to the struct hierarchy shown below.

with the shortest length is retained. This can happen when different branches in a given function execute different return statements. Using the shortest return hierarchy is a simple heuristic that maximizes the number of possible full matches for values assigned to function returns.

For global variable searches, we must be able to propagate the return hierarchies of the different functions that reference the global variable being searched. We accomplish this using a function return map, which maps a function name and its source file path to the respective return struct hierarchy and output parameter list. After all references to the global variable are parsed, we iterate through each function in the map, obtain a list of references to the given function, and filter out references that are not function calls. We then first consider the return struct hierarchy of the function. If a reference is an assignment, we search the assigned variable in the caller in the same way as described in section 3.1.2.1. We then handle the output parameters of the respective callee in the same way described in 3.1.2.2 by matching each parameter with the respective output arguments of the given caller. If there is a full match in the call chain of one of the caller functions, the respective caller is added as an entry point in the call graph. Each time we evaluate a function return map, we create a new function return map for the return values of the caller function. This process is repeated recursively until the function return map created in a given step is empty.

Since macro functions are treated as regular functions, we consider their return statement to be the last expression in the given macro function. Since an expression may span multiple lines of code, we must keep track of the range of line numbers spanned by the return statement of a given macro. When evaluating references within macro functions, we must always check if the given expression is within this range of lines and treat it as a normal return expression if that is the case.

3.1.3 Call graph expansion

The intermediate call graph obtained in the value propagation phase is expanded by determining the complete set of callers of the entry point functions. This is done by looking up the references to a given entry point and inserting a node for every caller, with an edge going from the entry point to a caller. This process is repeated recursively for each entry point caller until either the top of the call chain or a specified depth limit is reached. Common function names used in the test programs included in the Linux kernel source code, such as `main`, `init`, and `test`, are ignored to prevent state explosion. Special macro function, such as `module_init`, `module_exit` and `MODULE_LICENSE` are also ignored. This phase produces the final call graph that will be used in the dynamic detection of the usage of sysctl knobs.

3.2 Dynamic Detection

The detection of the usage of sysctl knobs at runtime is performed by an eBPF program that hooks both the root and entry point functions in the call graphs of the knobs that are being considered. This program is generated by reading the call graphs obtained through the static analysis step for each knob. Since functions in the call graphs may have been inlined by the compiler or may not be probed, certain root or entry point functions must be replaced by their respective parents.

For each sysctl knob, we first identify the respective set of root functions and entry points in their call graph. For each function, we check if a corresponding entry exists in `/proc/kallsyms`; if not, we assign the callers of the function in question as the respective root or entry point. In addition, some functions are only used during the boot process of the kernel and are freed from memory afterward. These functions are identified by the `_init` label in the kernel code. While these functions have a corresponding entry in `/proc/kallsyms`, they cannot be hooked since they do not exist after the kernel has finished booting up. As a result, these functions are also removed from the set of roots and entry points being hooked in the same way as the functions not present in `/proc/kallsyms`. After the relevant set of functions is identified, we then proceed by identifying the set of root functions that can be reached from each entry point in each knob call graph.

For each root function identified, we traverse the call graph for each knob until either an entry point or the top of the respective call chain is reached. A root function can have no corresponding entry point that can be hooked. This phenomenon happens when none of the corresponding entry point function callers are valid functions with a respective entry in `/proc/kallsyms`. It is also possible for a function to be both an entry point and a root. In this case, we consider the root function in question to be its own entry point. After identifying the respective entry points, we generate the eBPF program.

Each entry point function has a corresponding flag in the eBPF program that is set when the function is called and cleared when the function returns, and the corresponding root functions check this flag to determine if a given knob has been accessed. To accomplish this, we create both a `kprobe` and `kretprobe` hook point for each entry point so that the corresponding flag for the given entry point is set when the `kprobe` hook point is hit and cleared when the `kretprobe` hook point is hit. The flags for the entry points are stored in a BPF per-cpu array map, with each entry point assigned a corresponding index. The root functions are hooked using a `kprobe` hook point that checks the flags for the corresponding entry point for each corresponding knob and prints a message if the corresponding flag is set. This message contains the name of the knob and the respective root and entry point that was hit. As mentioned earlier, a function can be both an entry point and a root for a given knob. In this case, the root function will always print a message for the respective knob. The same is done for root functions that have no valid entry points. A single message is printed per knob per hit, meaning that the number of times a knob was accessed can be determined by checking the number of corresponding messages printed.

Chapter 4

Implementation

This section will explain some of the implementation details of our static analysis. Section 4.1 explains the strategy used to extract the complete expression to which a cscope query result refers. Section 4.2 describes how we determine the variable in which a given sysctl knob is stored before we start the value propagation step of the static analysis. Section 4.3 explains how we determine if a value is local to a function and how we determine which query results are within a given function. Section 4.4 explains how we find the function declaration for a given function. Finally, section 4.5 describes the database we use to decrease the latency of our static analysis process.

4.1 Extracting Complete Expressions

Since expressions in C can span multiple lines and each cscope query result only contains one line of code, we must determine if the expression in the query result is complete. We must be conservative in our estimate to ensure we always evaluate complete expressions. As such, the only types of expressions that can be considered complete are assignments, control flow expressions, `return` statements, and `case` statements that end in a colon (:). Furthermore, a complete expression must not have any mismatched parentheses and must end in either a semicolon (;), a close bracket (}) or, if it is a case statement, a colon (:). An assignment statement to an array field that is a part of a compound literal, starting with a dot (.) and ending with a comma (,), is also considered complete. Finally, a complete expression must start with a character that can be a part of a valid variable name, a star (*), or a dot (.). Expressions that can be determined to be a part of a macro definition, meaning that they either start with a number sign (#) or end with a backslash (\), are never considered complete. If an expression is not considered complete, we must search the respective source file to extract the corresponding complete expression.

A complete expression can be extracted from a cscope query result by searching the respective source file and finding the first complete expression that ends in a line number greater than or equal to the line number in the query result. To accomplish this, we concatenate successive lines in the source code file until a stop condition is reached. If this condition is reached before we reach the line number in the cscope query result, we clear the buffer used to store the concatenated lines and start checking the next expression. Different types of expressions have different types of stop conditions.

Stop conditions are determined by checking the current line's contents. Furthermore, complete expressions must not contain mismatched parentheses unless otherwise specified. The simplest stop condition is achieved when the last character in the current line is a close bracket (}). Another possible stop condition is fulfilled when the last character in a line is a semicolon (;) and there are no mismatched parentheses. One exception occurs when evaluating struct declarations, where the stop condition is only reached when the declaration is closed by a close bracket (}). Another example of a general stop condition is detected when the current line ends with an open bracket `{` and we are not currently inside a compound literal. In addition to those general conditions, some statements have specific stop conditions.

We must account for the particularities of certain types of statements when checking for stop conditions. One example occurs when evaluating `case` statements or `goto` labels. In those cases, the stop condition is fulfilled when the last character in the current line is a colon (:). Another example that must be considered occurs when evaluating control flow expressions with a single line in their bodies. Since those expressions end in neither an open bracket nor a semicolon, the stop condition is achieved when there are no mismatched parentheses. We also consider to be a special case an assignment to a struct field as a part of a compound literal. In this case, we consider the assignment to be complete by itself, meaning that the stop condition is reached when the first character of the current expression is a dot, the last character of the current line is a comma, and we are not inside a compound literal that is a part of the assignment to the struct field. In addition to those particular cases, preprocessor directives must be carefully considered.

Since the C preprocessor has a different syntax from the C language, preprocessor macros must have their own stop conditions. One possible stop condition is fulfilled when the last line ends with a backslash, and the current line does not. In this case, this stop condition is achieved even if mismatched parentheses exist in the current expression, which occurs when defining macro functions with multiple statements in their bodies. Another possible stop condition occurs when the current line has a preprocessor directive and no mismatched parentheses. To ensure that these stop conditions are properly detected, lines of code must be properly sanitized before they are analyzed.

Lines of code must be properly cleared from comments or string literals before they are analyzed. When checking for the presence of comments in a line, we must check if the symbols that signify the start of a comment, which are either // or /*, are not inside a string literal. This is one example of how string literals can interfere with the parsing of codes of line, which is why they must be removed before a line of code is parsed. One special case of comments that must be considered is the usage of the `#if 0` preprocessor directive. In this case, even though the body of the directive is not a part of the code and can be considered a comment, cscope query results might still reference code inside this type of directive. If this happens, we must return an empty string as the complete expression to ensure this non-valid code is not evaluated. With this process in place, we can ensure that we only evaluate valid code.

We must keep track of the number of open brackets to determine if the current line of code is part of a compound literal. This process can be complicated by the presence of preprocessor directives used for conditional compilation, as the number of open brackets inside the body of different conditional directive blocks can be inconsistent. To handle that, when entering a conditional preprocessor block, we must keep track of the number of open brackets before entering the block, and when exiting the block, we must keep track of the number of bracket blocks that were opened inside the conditional preprocessor directive block. With this, if we encounter a conditional block with the same condition as another block that was parsed earlier, we can ensure that we remember how many open brackets must be closed to ensure that the overall number of open brackets is consistent. To achieve this, we define a bracket state tree that contains a node for each preprocessor directive condition encountered. When entering a new conditional block, we save the number of open brackets of the current block and compute the difference between the current number of open brackets and the number of open brackets when the current block was entered. When leaving a `#if` or `#elif` conditional block, we save the current number of open brackets and the number of bracket blocks stored in the current block and restore the number of open bracket blocks before the current block is entered. When exiting `#else` preprocessor blocks, we keep the current number of open brackets unchanged. The preprocessor directives themselves are removed from the lines of code being parsed. This procedure ensures that the number of open brackets is always consistent.

4.2 Finding Sysctl Knob Variables

In the simple case, the `.data` field of a given knob is set to the name of the global variable that stores the value of the knob, as explained in section 2.1. If this is the case, we simply extract the same of the variable and struct hierarchy from the `.data` field and perform a global search. For some knobs, however, the `.data` field of the respective `ctl_table` is either omitted or set to NULL, and, as a result, we must find another way to find the name of the respective variable.

Some `ctl_table` entries define a custom handler in their `.proc_handler` field instead of setting the `.data` field to the variable in which the value of the knob is stored. In this case, we know that the buffer used to read and write to the sysctl knob from user mode corresponds to the `buffer` parameter in the handler function. As such, we can perform a local variable search for that parameter within the handler and identify the name of the variable in which this value is stored. Handlers usually copy the `ctl_table` entry to a local variable and set the respective `.data` field to another local variable and call one of the generic handler functions to read and write to the variable. In this case, we can find the function call of the handler in which the buffer is being passed as an argument and find the name of the local variable used to store the `ctl_table` entry by looking at the function call's arguments. From there, we can search for assignments to the data field of the `ctl_table` variable and extract the name of the local variable storing the value of the knob. We can then perform a local variable search within the custom handler function. Sometimes, handlers will allocate a new buffer for the `.data` field and use it directly. We can identify this by checking for an assignment of the data field to the value returned by the `kmalloc` function. In this case, we do a local search for the table entry variable with a struct hierarchy of `{data}`. Some knobs may omit the `.data` field set a generic handler function in their `.proc_handler` field.

At times, the value of the pointer stored in the `.data` field of a `ctl_table` may be set at runtime, and a generic handler is used in the respective `.proc_handler` field. In this case, we can perform a global variable lookup for the sysctl table where the respective `ctl_table` is stored. From there, we can identify accesses to a specific `ctl_table` entry by checking for array accesses to the index of the `ctl_table` entry within the sysctl table array. We can then identify the pointer being assigned to the `.data` field and search for the respective variable.

4.3 Filtering References to Variables

When searching for a given variable, we must first determine if it is global or local, and, in the latter case, we must filter out all query results that are not in the function in which the variable was declared. As explained in 3.1.2.1, determining the locality of the variable is done by checking if a variable declaration exists in either the body or arguments of a function. The function to which a given query result refers is usually shown in the query result itself; however, due to a bug in cscope, it may not be reported correctly. The next subsections will explain in detail how these two tasks are accomplished.

4.3.1 Detecting Variable Declarations

There are multiple ways to declare variables or functions in C. Figure 4.1 shows examples of variable declarations currently supported. We will go into detail on how each type of declaration is detected.

```
int a;
struct s* b;
int func();
int func(int a, char* b);
int a , b, c;
char *a = NULL, *b = NULL; *c;
#define func(a, b) a + b
typeof(a) b;
struct s {int a; int b;} d;
struct {int a; int b} d;
```

Figure 4.1: Examples of valid variable declarations in C.

The first two cases show standard variable declarations comprising at least two tokens separated by spaces. In this case, a token is considered to be a string that only contains characters that can be present on valid variable names. As such, we can determine that a statement is a variable declaration for `a` if it contains at least one token followed by the token `a`. Structs and unions must be considered carefully, as we must be able to distinguish between a declaration of a struct or union type and a declaration of a variable of the given type. As such, when evaluating those expressions, the tokens `struct` and `union` are not counted in the number of tokens in an expression. Furthermore, to be able to distinguish between a type name and the name of a variable itself, we must also ignore type qualifiers, such as `const`, `volatile`, and `register`, when counting the number of tokens in an expression. In addition, we must also say that if certain keywords, such as `case`, `return`, or `do`, are present in the current statement, then it is not a variable declaration. A declaration to a function or an argument of a function can be detected in the same manner since it follows a similar structure, as seen in the third and fourth examples in figure 4.1. When examining the number of consecutive tokens, we must, however, carefully consider the presence of some special characters.

We must consider the presence of a start (*) carefully, as it may be a part of a variable declaration or an operator. To account for that, for variable declarations, the presence of a (*) in an expression that either does not contain an assignment or is on the left-hand side of an expression does not break the flow of valid consecutive tokens. For function parameter declarations, we say that if there are at least two consecutive tokens before the parenthesis, which indicates a valid function declaration, then the presence of a start does not break the flow of consecutive tokens for the parameter declarations. Different types of function declarations, however, force us to open a few exceptions to our previously described rules.

The fifth example in figure 4.1 shows the declaration of multiple variables in the same line. In this case, this can be considered to be a valid declaration to variable `b` even though the respective token is not immediately followed by at least one token. As such, we must also say that an expression is valid if it contains at least two consecutive tokens and the variable's name in question is followed by a comma (,). As seen in the sixth example, multiple variables can be assigned and declared in the same line. This also introduces some complications with the star (*) character, as we must be able to determine if the star is a part of a variable declaration or an operator in a function call. As such, if a star immediately follows a comma and does not come after an open parenthesis in an expression, it can be ignored, and the token after it can be considered to immediately follow a comma. There are a few other special characters that must be handled differently.

Since macro functions are treated as normal functions, we must be able to detect the macro declaration shown in the seventh example as a declaration of both the macro function itself and a declaration of its arguments. As such, we must also consider the preprocessor directive `#define` as a valid token. Furthermore, even though the argument names in the declaration do not follow another token, they must be detected as variable declarations. To account for this, if the current statement contains the `#define` token, and a token is within the parentheses that determine the argument declaration of the macro function, then this is a variable declaration for the given token.

In addition, variables declared inside macros often use the `typeof` construct in their declaration, as shown in the eighth example. As such, we must count the entire `typeof` expression, including the parenthesis, as a single valid token.

The ninth and tenth examples in figure 4.1 show two valid ways of declaring a struct variable. To account for those, if an expression contains either the `struct` or `union` keywords and contains an expression within brackets, we must count the entire expression, including the brackets, as a single valid token. As a result of that, we must make sure that, when extracting complete expressions as described in section 4.1, the entire expression within brackets of a struct declaration must be included in the respective complete expression.

4.3.2 Determening The Locality of Query Results

The results of queries returned by cscope include the name of the function in which the respective references are found; however, due to a bug in cscope, the name of the function included in the query results will sometimes be incorrect. When this happens, all query results found within a given function will be incorrectly classified as part of another function. As such, if we are checking for references to a variable within a function and none of the references are a part of the given function, we must perform an additional step. When resolving function declarations, as it will be explained in 4.4, we also extract the first line of the function declaration. Then, if there are no query results for a variable within a function, we determine the last line of the body of the given function. This process is done by checking the number of open brackets within the source file starting from the first line of the function declaration. Then, we collect all references that are a part of the same source file as the function. For each of them, we check if the reported line number is within the range of lines spanned by the given function and consider the respective reference to be a part of the function if that is the case. This workaround allows us to identify the references of a variable within a function even when the name of the query results is not reported correctly.

4.4 Finding Function Declarations

When searching for the declarations for a given function, we must be able to differentiate between function declarations and function calls and determine which declarations are in the same scope as the respective function call. While cscope can specifically search for global declarations, some static function declarations will not show up in the query results if this feature is used. As such, we must search for all references to a function and determine which of the query results refers to the declaration of the function. We use the procedure described in section 4.3.1 to identify function declarations. We further filter out the function prototypes from our search and only consider the actual function declarations. There might be multiple functions with the same name, so we must determine which function declarations are in the same scope as the function call being analyzed. To accomplish this, we use a simple heuristic in which we assume that static functions or function macros defined in a `.c` source file can only be called within the same source file. In addition, we also determine that if a function declaration exists in the same source file as the function call, then the given declaration will always be the one to which the respective call refers. This system can distinguish between different functions with the same name defined in different source files.

We currently do not support functions with a variable number of arguments. As such, declarations with ellipses (...) in their parameter declarations are ignored, and the respective function calls are not analyzed. Since such functions are often debugging functions, these are most likely not relevant to our static analysis.

4.5 Database

We created a database to cache some data to decrease the overall latency incurred by continuously performing cscope queries and the file I/O to extract complete expressions. This database comprises three tables: the function-value query table, the function declaration table, and the macro return ranges table. These tables are kept in memory and written to a file on disk, each stored in a separate file. Before starting the static analysis, these three tables are read from disk and loaded into memory. While performing the static analysis, the in-memory and the persistent copies of the three tables are updated continuously.

4.5.1 Function-Value Query Table

This table maps a key containing the source file, function, and name of a given variable to the query results obtained from cscope. Since query results often do not contain complete expressions as they may span multiple lines, the complete expressions for each query result are also stored in the table. Global variables have a special token for the function name and source file part of the key, and queries for functions have a special token for the variable name. If a given variable has already been queried, the results can be retrieved from the in-memory table instead of repeating a cscope query. A given entry in the table is committed to disk after the variable search for the given variable has been completed and all query results have been parsed.

4.5.2 Function Declaration Table

This table maps a key with a function name and its source file to the complete function declaration and the line number in the source file in which the respective declaration starts. These function declarations are usually obtained while resolving function calls, as described in section 3.1.2.2. This entry can also be obtained when checking the declaration of the current function to check for assignments for output parameters, as described in section 3.1.2.1. The entries of this table are added to the in-memory structure and committed to disk right after the respective function declaration has been resolved.

4.5.3 Macro Return Ranges Table

This table maps a given function macro name and its respective source file path to the range of line numbers spanned by the last expression of the macro function. This information is needed to determine if an expression in a macro function is a return statement, as described in section 3.1.2.3. These ranges are obtained while resolving the complete expression for a query result that includes one of the lines spanned by the last expression in a macro function. The respective entries are added to the in-memory structure and committed to disk right after the corresponding complete expression is obtained.

Chapter 5

Evaluation

5.1 Results

For our evaluation, we manually determined the knobs that were relevant for the benchmarks that were used for our testing and classified them according to the subsystem that they regulate. In total, we considered 241 different knobs, with the different categories and the number of knobs in each category being shown in table 5.1.

Knob classification	Number of knobs
TCP	98
UDP	7
IP	13
Network Device Management	28
Scheduling	11
Virtual Memory	38
Memory Management	6
Interprocess Communication (IPC)	4
Filesystem	7
Logging/Tracing	25
BPF	3
Locking	1

Figure 5.1: Classification of the knobs tested in the evaluation.

5.1.1 Static Analysis

Our static analyzer was tested on the knobs described in figure 5.1 using the source code of version 5.15.152 of the Linux kernel. Some parts of the Linux kernel source code that were irrelevant to our analysis were removed to reduce our search space and ensure the process ran smoothly. In particular, we excluded testing tools, drivers, and architecture-specific codes for all architectures except x86. In addition, C header files containing assembly macros were also removed as these cause issues with our static analysis process. Knobs were analyzed one after the other, and the databases described in section 4.5 were built iteratively, with the entries added during the static analysis of one knob being used for subsequent knobs.

Out of the 241 sysctl knobs searched, we were able to identify the respective variables in the Linux kernel for 201 of them. Out of those 201, we could detect references and build a call graph for 118 of them. The classification of the latter knobs is shown in figure 5.2. The resulting call graphs were then used to test the dynamic detection phase.

Knob classification	Number of knobs
TCP	11
UDP	5
IP	7
Network Device Management	18
Scheduling	10
Virtual Memory	32
Memory Management	5
Interprocess Communication (IPC)	3
Filesystem	5
Logging/Tracing	19
BPF	3
Locking	0

Figure 5.2: Classification of the knobs with detected references.

5.1.2 Dynamic Detection

We evaluated the dynamic detection of the usage of sysctl knobs on two different applications. In particular, we tested Memcached [4] and Redis [12], both running the Memtier benchmark [11]. These benchmarks were run on a QEMU virtual machine running version 5.15.152 of the Linux kernel on an Intel Xeon Platinum 8468 cluster with two NUMA nodes, with 48 CPUS in each socket and 512 GB of RAM.

For each benchmark, we first obtained the respective kernel call graph using perf with the `-a` flag and used this call graph to select a list of knobs that were potentially accessed by the benchmark. We accomplished this by searching the call graph produced in our static analysis step and comparing it with the perf call graph. We searched each function in the call graph, starting from the roots, and, if any of the functions in the call graph of a given knob were recorded in the perf call graph of a particular benchmark, we considered that knob to be potentially accessed. We then collected the call graphs of all potentially accessed knobs and used them to test our dynamic detection.

The call graphs of potentially used knobs for each benchmark were used to generate an eBPF program that was loaded while the benchmarks were run. When the eBPF program detected the potential usage of a given knob, a message was printed in the Linux trace pipe. Only the trace events that correspond to the processes run by a given benchmark were considered. Since certain functions in the call graph may be accessed very often, the large number of printed messages may result in lost tracing events. Each benchmark was run multiple times to account for this, and the respective eBPF program was modified for each iteration. In particular, in a given run, the message printed for a given entry point-root pair for a given knob was recorded, and the corresponding print statement was removed from the eBPF program used in subsequent runs. This process was repeated until no more messages were printed in the trace pipe. The list of knobs access across all runs for each benchmark was then recorded.

For Memcached, we were able to detect accesses to a total of 35 knobs. For Redis, however, we were only able to detect one knob, namely, the `vm.mmap_min_addr`, which is a part of the virtual memory category in our classification. The number of knobs detected in each subsystem for Memcached is shown in figure 5.3.

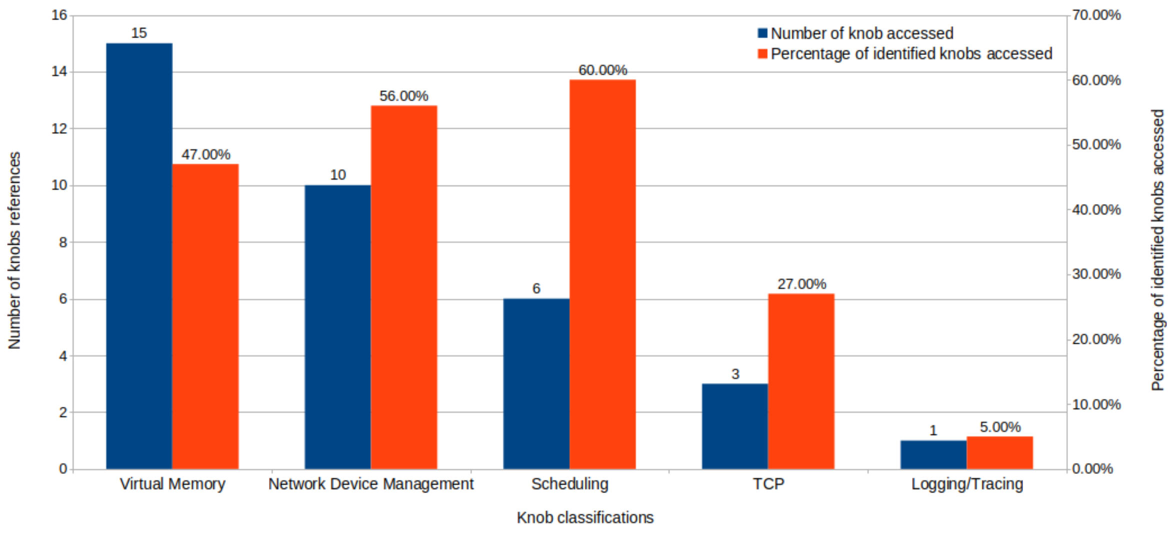

Figure 5.3: Number of knobs accessed by Memcached in each classification category and the respective percentage of knobs with call graphs represented by these numbers.

5.2 Discussion

Our static analyzer was not able to locate the respective variables for 40 of the knobs that were tested. Out of those, 24 of them have no corresponding `ctl_table` entries and no references to them in the source code of the Linux kernel version used in our testing. These knobs seem only to have been introduced in versions of the kernel that came after the source code of the version that was tested. In addition to those, 13 of the knobs that could not be located do have corresponding `ctl_table` entries, but the respective entries are never read within the Linux kernel. All of those knobs are a part of the Netfilter Conntrack subsystem and are all related to TCP. This leaves us with a total of 3 sysctl knobs that are used in the Linux kernel but could not be located by our static analyzer. These knobs are accessed in ways that are not accounted for the steps described in section 4.2 and could not be implemented within our system due to time constraints.

There were a total of 83 knobs that could be located, but no references to their specific struct hierarchy were found. Most of these knobs were stored in the field of the `init_net` struct variable that is used to initialize the network interfaces in the Linux kernel. One possible reason for this is that our static analyzer is currently unable to parse function pointer assignments. As a result of that, calls to function pointers stored in global variables are not considered, and the respective knob values are not propagated through the function call. This is indeed a limitation of our tool that can also be observed for other knobs. In particular, the entry point function for the net.ipv4.ip_forward knob, `devinet_init_net`, which assigns the value of the knob to one of its return parameters, is never called directly by any other function in the kernel and is instead only called through a pointer stored in a global variable. As a result of that, the static analyzer is not able to propagate the value of the given knob through any of the functions that call `devinet_init_net` and use the corresponding output argument. Currently, our tool is only able to resolve calls to function pointers that were passed as arguments to function calls, and the proper framework for resolving function pointer assignments could not be implemented due to time constraints.

Another limitation of the static analyzer is that it does not account for the indirect effects of the value of a knob. One example of this is the `kernel.numa_balancing` knob, which is used as the condition of an if statement that determines the value of another variable. Since the value of the knob is not directly assigned to this other variable, this indirect effect of the knob is not accounted for. This phenomenon could be accounted for by parsing all variable assignments and function calls in conditional branches if a value of a knob is used to determine whether or not a branch is executed.

Although our dynamic detection routine managed to identify references to several different knobs for Memcached, it failed to do so for Redis. One of the reasons for this is the fact Redis performs significantly fewer system calls than Memcached, and, as a result of that, the number of potentially used knobs is lower. In addition, most of the system calls performed by Redis are TCP-related, and, as explained earlier, our static analyzer failed to identify references to many networking-related knobs, particularly the ones that involve TCP. As a result of that, many of the knobs potentially used by Redis go undetected.

Currently, our method of detecting the usage of sysctl knobs at runtime is prone to false positives. Only taking into account the entry point and root functions in the call graphs of a knob is not sufficient to detect with certainty that a given knob was accessed. One possible design that would eliminate some of the false positives inherent to our current method used for the dynamic detection of the usage of the knobs would be to check the current stack trace when a kprobe hook point is hit for one of the root functions. To accomplish this, we would merge the call graphs for all of the knobs into a single call graph that would be stored in a constant data structure in our eBPF program. Then, for each function in the call graph, we would store the list of knobs in which the given function is a part of the respective call graph. We would then extract the address of each of the functions in `/proc/kallsyms` and add it to the respective nodes in our unified call graph. In addition, since the return addresses in a stack trace do not correspond to the address of the start of a function, we could estimate the range of addresses spanned by the code of a given function by using `/proc/kallsyms` to determine the address of the next function in memory. With that, whenever the hook point for a root function is hit, we would attempt to match the current stack trace with one of the possible call chains in the unified call graph by checking if each corresponding return address is within the range of addresses spanned by the code of each successive function in a given call chain. We would then check the list of knobs at each step of the call chain to determine the set of knobs that could have been potentially accessed. This method of attempting to match the stack trace of a given root function with the edge of the call graph was not implemented in our dynamic detection routine due to time constraints.

Chapter 6

Conclusion

Our static analyzer was able to locate 201 sysctl knobs and was able to build call graphs for 118 of them. Out of the 40 knobs that could not be located, only 3 of them are used in the Linux kernel code we analyzed. Currently, our static analyzer does not fully support parsing function calls for function pointers, and, as a result of that, it was not able to detect references to 83 of the knobs it was able to locate. In addition, knobs that are not used directly and are instead used to determine the control flow of the code are also not fully accounted for. These issues will be addressed in further developments of this project.

Our dynamic detector was able to detect references to 35 knobs for Memcached and 1 knob for Redis. The lack of networking-related knobs being detected is mainly due to the limitations of the static analyzer that were described earlier. While our current dynamic detection routine is complete and can detect all references described in the call graphs, the method used to detect references to knobs is prone to false positives. This issue happens because our method does not distinguish between different possible call chains between an entry point and a root function. This limitation can be fixed by checking the stack trace when a root function is executed and attempting to match it with one of the edges of the call graph. We will address these issues in further revisions to this project.

Bibliography

[1] Hans-Bernhard Bröker, Neil Horman, and Joe Steffen. *Manpage of CSCOPE*. Mar. 13, 2002. URL: `https://cscope.sourceforge.net/cscope_man_page.html` (visited on 06/19/2024).

[2] Jonathan Corbet. *Managing sysctl knobs with BPF*. Apr. 9, 2019. URL: `https://lwn.net/Articles/785263/` (visited on 06/20/2024).

[3] Shen Feng, Jorge Nerin, Terrehon Bowden, and Bodo Bauer. *Documentation for /proc/sys/net/*. 2009. URL: `https://docs.kernel.org/admin-guide/sysctl/net.html` (visited on 06/19/2024).

[4] Brad Fitzpatrick. *memcached - a distributed memory object caching system*. URL: `https://memcached.org/` (visited on 06/20/2024).

[5] Matt Fleming. *A thorough introduction to eBPF*. Dec. 2, 2017. URL: `https://lwn.net/Articles/740157/` (visited on 06/19/2024).

[6] eBPF Foundation. *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: `https://ebpf.io/what-is-ebpf/` (visited on 06/19/2024).

[7] Bernhard Heinloth, Marco Ammon, Dustin T. Nguyen, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. "Cocoon: Custom-Fitted Kernel Compiled on Demand". In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. SOSP '19: ACM SIGOPS 27th Symposium on Operating Systems Principles. Huntsville ON Canada: ACM, Oct. 27, 2019, pp. 1–7. ISBN: 978-1-4503-7017-2. DOI: `10.1145/3365137.3365398`. URL: `https://dl.acm.org/doi/10.1145/3365137.3365398` (visited on 06/20/2024).

[8] Red Hat Inc. *BPF_MAP_TYPE_ARRAY and BPF_MAP_TYPE_PERCPU_ARRAY — The Linux Kernel documentation*. 2022. URL: `https://docs.kernel.org/bpf/map_array.html` (visited on 06/19/2024).

[9] Alexander Jung, Hugo Lefeuvre, Charalampos Rotsos, Pierre Olivier, Daniel Oñoro-Rubio, Felipe Huici, and Mathias Niepert. "Wayfinder: towards automatically deriving optimal OS configurations". In: *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '21: 12th ACM SIGOPS Asia-Pacific Workshop on Systems. Hong Kong China: ACM, Aug. 24, 2021, pp. 115–122. ISBN: 978-1-4503-8698-2. DOI: `10.1145/3476886.3477506`. URL: `https://dl.acm.org/doi/10.1145/3476886.3477506` (visited on 06/20/2024).

[10] Jim Keniston, Panchamukhi Panchamukhi, and Masami Hiramatsu. *Kernel Probes (Kprobes) — The Linux Kernel documentation.* URL: `https://docs.kernel.org/trace/kprobes.html` (visited on 06/19/2024).

[11] Redis Labs. *Memtier Benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool.* URL: `https://github.com/RedisLabs/memtier_benchmark?tab=readme-ov-file` (visited on 06/20/2024).

[12] Redis Labs. *Redis - The Real-time Data Platform.* URL: `https://redis.io/` (visited on 07/04/2024).

[13] Keith Owens. *kallsyms(8) - Linux manual page.* Jan. 31, 2002. URL: `https://www.unix.com/man-page/redhat/8/kallsyms/` (visited on 06/19/2024).

[14] The Linux Documentation Project. *The __init and __exit Macros.* Aug. 5, 2006. URL: `https://tldp.org/LDP/lkmpg/2.4/html/x281.htm` (visited on 06/19/2024).

[15] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. "An analysis of performance evolution of Linux's core operations". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* SOSP '19: ACM SIGOPS 27th Symposium on Operating Systems Principles. Huntsville Ontario Canada: ACM, Oct. 27, 2019, pp. 554–569. ISBN: 978-1-4503-6873-5. DOI: `10.1145/3341301.3359640`. URL: `https://dl.acm.org/doi/10.1145/3341301.3359640` (visited on 06/20/2024).

[16] Mike Shaver, Eric Dumazet, and David Miller. *Linux kernel source code at net/core/sysctl_net_core.c.* URL: `https://github.com/torvalds/linux/blob/e5b3efbe1ab1793bb49ae07d56d0973267e65112/net/core/sysctl_net_core.c#L560` (visited on 06/20/2024).

[17] George Staikos. *sysctl(8) - Linux manual page.* Aug. 19, 2023. URL: `https://man7.org/linux/man-pages/man8/sysctl.8.html` (visited on 06/19/2024).

[18] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem". In: *Proceedings of the sixth conference on Computer systems.* EuroSys '11: Sixth EuroSys Conference 2011. Salzburg Austria: ACM, Apr. 10, 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: `10.1145/1966445.1966451`. URL: `https://dl.acm.org/doi/10.1145/1966445.1966451` (visited on 06/20/2024).

[19] Linus Torvalds, Thomas Weißschuh, and Joel Andres Granados. *Linux kernel source code at include/linux/sysctl.h.* URL: `https://github.com/torvalds/linux/blob/92e5605a199efbaee59fb19e15d6cc2103a04ec2/include/linux/sysctl.h` (visited on 06/19/2024).

[20] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. "OCOLOS: Online COde Layout OptimizationS". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).* 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO). Chicago, IL, USA: IEEE, Oct. 2022, pp. 530–

545. ISBN: 978-1-66546-272-3. DOI: 10.1109/MICRO56248.2022.00045. URL: https://ieeexplore.ieee.org/document/9923868/ (visited on 06/20/2024).