Better Program Analysis for Security via Data Flow Tracking and Symbolic Execution

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Navid Emamdoost Balajorshari

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stephen McCamant, Adviser

August 2021

© Navid Emamdoost Balajorshari 2021

Acknowledgements

I would like to thank my adviser Stephen McCamant for his continuous support and supervision during my eight years at the University of Minnesota. He was always available to discuss openly and share valuable insights and give me constructive suggestions. I learned a lot from him. I also thank Kangjie Lu for his guidance and collaboration on the second project of this thesis. Additionally, I thank Nick Hopper and John Sartori for accepting to be on my thesis committee and providing ideas towards improving it. I also thank my family for their unconditional love and support through the years. I am grateful to friends who made living abroad a more pleasant experience. I am thankful to my co-authors and lab-mates for assisting me through different projects and papers. Last but not least, I acknowledge all my teachers and professors who gave the best of themselves over all the years of my studies.

Dedication

To my parents: Mahasti & Mehrdad.

Abstract

Program analysis techniques have numerous applications in software optimization and correctness. Focusing on software security, analyses like data flow analysis and symbolic execution have been proven effective in many settings. Information-flow security measures and bounds the propagation of sensitive information throughout the code. Automatic vulnerability detection provides tooling for tracking and identifying potentially buggy code. Data flow analysis is scalable but imprecise due to over-approximations. Symbolic execution is precise and sound but suffers on scalability. In this thesis, we develop multiple dynamic and static analysis techniques that employ data flow analysis and symbolic execution together to address imprecision and scalability issues.

More specifically, we revisit quantitative information flow analysis where the goal is measuring the amount of information flow from source to sink. We propose techniques to enable the incorporation of symbolic influence measurement and improve the precision of the final result. For static vulnerability detection, we propose a new tool to effectively detect memory leak bugs in big codebases like an OS kernel even in specialized modules. Our tool automatically identifies allocation and deallocation functions, and then reasons about the true location of memory release. It also employs under-constrained symbolic execution to improve the true positive ratio. We were able to detect numerous new memory leak bugs in the Linux kernel code. In both scenarios, we employ data flow analysis to shrink the search space for symbolic execution in a way that still can benefit from its precise property checking.

Contents

List of Tables vii

List of Figures viii

1 Introduction 1

2 Background and Related Work 4

2.1 Data Flow Analysis 4

2.2 Symbolic Execution 6

2.3 Static Vulnerability Detection 11

3 Quantitative Information Flow Analysis 14

3.1 Information Flow as Network Capacity 15

3.1.1 Constructing the Network of Flow 16

3.1.2 Calculating the Maximum Flow 17

3.2 Symbolic Execution for Better Precision 17

3.2.1 Internals of FCFB 20

3.2.2 Region Exploration 22

3.3 Case Studies 25

3.3.1 Count Vowels Binary 25

3.3.2 De-saturate Image Transformation Binary 29

3.3.3 cksum Binary (coreutils) 30

4 Static Vulnerability Detection 34

4.1 A Study of Kernel Memory Allocation 34
4.2 Memory Leak 37
4.3 Overview of K-MeLD 42
4.4 Allocation and Deallocation Identification 45
4.4.1 Identifying Allocators 45
4.4.2 Context-aware Rule Mining for Deallocation Identification . . 47
4.5 Ownership Reasoning 49
4.5.1 Enhanced Escape Analysis 50
4.5.2 Consumer Function Detection 52
4.6 Detecting Bugs Using Mined Rules 55
4.6.1 Under-constrained Symbolic Execution 56
4.7 Implementation 57
4.7.1 Potential Allocation Functions 57
4.7.2 Path Exploration 58
4.7.3 Context-aware Rule Mining 59
4.7.4 Ownership Reasoning 60
4.7.5 Pattern Matching 61
4.7.6 Under-constrained Symbolic Execution 61
4.8 Evaluation on the Linux Kernel 63
4.8.1 Set of Allocations and Associated Deallocations 63
4.8.2 Found Bugs 64
4.8.3 False Positive Analysis 64
4.8.4 False Negative Analysis 65

5 Conclusion and Future Work 68

5.1 Quantitative Information Flow Analysis 68

5.2 Static Vulnerability Detection 69

References **71**

List of Tables

3.1 Neighbor regions of (i, j) with a step size of ss 23

4.1 False Negative Analysis results based on previous memory Leak CVEs 66

List of Figures

2.1 Example of Symbolic Execution 7

3.1 An example of overestimation by FlowCheck 18

3.2 Main Part of Count Vowel Code 26

3.3 FCFB region exploration for `CVB` 27

3.4 Main Part of De-saturate Transform Code 28

3.5 Input PPM Image 30

3.6 FCFB region exploration for `de-saturate` binary 31

3.7 Main Part of CRC-32 computation of `cksum` 32

3.8 FCFB region exploration for `cksum` binary 33

4.1 A new memory leak bug (CVE-2019-19062) detected by K-MeLD: the path ending at line 25 needs to release `skb`. 38

4.2 Overview of K-MeLD. Identifying specialized allocation functions and the corresponding deallocatoin is followed by inter-procedural escape and consumer function analysis to determine the ownership of each allocation. Rule application checks the presence of deallocation and reports the bugs. 43

4.3 An example of escaping pointer: The allocation pointer `bounce_buf` is escaping via a pointer argument at line 33. But it is not enough to the prevent memory leak at line 27 (CVE-2019-19048). 51

4.4 Consumer function example: `t4_mgmt_tx()` consumes the buffer `skb` unconditionaly. 53
4.5 Conditional Consumer function example: `skb_put_padto()` consumes the buffer `skb` on failure. 54
4.6 An example of infeasible path introducing false positive. 56

Chapter 1

Introduction

Data flow analysis has been employed in compiler optimization for a long time [1]. It represents global data relationships in a program via static analysis methods to determine definition, use, and reach points of data in a control flow graph. Since then, data flow analysis gained numerous applications in program analysis specifically in security. By determining how data propagates through a program and where it is used, data flow analysis can help in identifying security issues like uninitialized use, resource leak, data leak, insecure argument to functions, etc. The type of data flow analysis mostly used in security is taint analysis. In taint analysis, a meta-data value is maintained for each value in the program to describe a special property of that value (for example if it is attacker controlled). The taint property is propagated through the program to other values whose computation is influenced by current tainted value. For example, an addition operation which takes a tainted value and an untainted one, produces a tainted result. Taint propagation is a binary property meaning that the result either becomes tainted or not.

Data flow analysis and as a result, taint analysis both are conservative so they may suffer from false positive reports. Pointer aliasing is a challenge to data flow analysis. It is determining if two pointers are pointing to the same memory location or not. Because of aliasing problem, false positives may occur in taint analysis.

Another threat to data flow analysis is control flow sensitive data propagation and complicated constraints leading to infeasible paths.

Symbolic execution is a program analysis technique to explore feasible paths in a program without actually running it. Instead of providing actual input, the inputs are symbols and execution proceeds as normal while the values may be symbolic formulas. The technique is used extensively in software verification and testing, as well as security. Symbolic execution can explore multiple paths by operating on symbolic variables. Such variables can represent different concrete inputs. This is useful in providing sound analysis for property checking. For each explored path a boolean formula is maintained that describes the satisfied conditions along the path, and a memory model maps variables to symbolic expressions or values. Each store updates the memory map, while a branch instruction updates the path formula. At each point a satisfiablity modulo theories (SMT) solver [5] can be used on the path formula to check for property violation along the explored paths, or just the feasibility of a path. Symbolic execution suffers on scalability. Path explosion is the main source of scalability issues in symbolic execution, where the number of candidate paths to explore may double at each branching point in the program. This leads to an exponential growth of paths to explore, affecting both processing time and memory requirements.

In this proposal, we combine data flow analysis with symbolic execution for security purposes. On one hand, symbolic execution can help data flow analysis by removing false positives and improving precision. On the other hand, data flow analysis can help reducing the search space for symbolic execution and improve its scalability. More specifically, we revisit two security domains and propose improved analysis techniques:

Information-flow Security. In information-flow security the goal is limiting the information propagation through the program. For example, a public output of the

program should reveal up to a threshold number of bits of information about its secret input. In Chapter 3 we utilize symbolic execution to improve the precision of quantitative information-flow analysis.

Static Vulnerability Detection. Programming languages like C that have little to no automatic memory management, are prone to memory errors. With an emphasis on resource release bugs, in Chapter 4 we propose a static vulnerability detection technique that utilizes multiple data flow analyses to detect a potentially buggy point in the code and then use symbolic execution to reduce false positives.

Chapter 2

Background and Related Work

In this chapter we cover background and related work on the two well-known and classic program analysis techniques: data flow analysis and symbolic execution, and their application in vulnerability detection.

2.1 Data Flow Analysis

Data flow analysis (DFA) determines how values of variables propagate throughout program execution. It is a flow-sensitive analysis meaning that it relies on control flow graph transitions to determine properties associated with each basic block. It defines algorithms to infer properties at each execution point. Over the past four decades, data flow analysis is extensively employed in compiler optimization, software testing, verification, and security.

For a given statement `s`, the program point just before executing `s` is `In(s)`, and the program point just after executing `s` is `Out(s)`. A forward analysis is a data flow analysis that tracks the propagation from `In` to `Out` for each statement. In contrast, a backward analysis is tracking the propagation from `Out` to `In` for each statement. The building block properties of DFA are *definition* and *use* statements. Every assignment is a *definition* of the variable assigned to. An statement that refers

to a defined variable is a *use* for that variable. A definition *reach*es a point if there exists a path from the definition to the point without re-definition. A re-definition *kill*s (overwrites) any previous definition. A *def-use* pair exists if there is at least one path from the definition *def* that reaches the *use*. An expression is *available* at a point if all the paths leading to the point contain a definition that reaches the point. There is no need to re-evaluate an available expression if it is stored somewhere. *Availability* analysis is a forward analysis that tracks the definitions from `In` to `Out` for each statement on all paths leading to a point. A variable is *live* at a point in the program if there exists a use on some paths originating from that point. Otherwise the variable is *dead*. There is no need to cache dead variables. *Liveness* analysis is a backward analysis that starts from each *use* and moves backward from `Out` to `In` until it reaches a *definition* on the path.

Static data flow analysis analyzes the code without actually executing it. It can cover all the available code but in some situations may need overapproximation to account for the cases that the state of program variable cannot be determined before execution[1]. Dynamic data flow analysis determines the variables propagation by executing the program. It may fail to cover all reachable code, but can be more precise in terms of which addresses of memory are accessed.

Static DFA has been used extensively in code optimization phase in compilers. Some classic techniques include redundant sub-expression elimination [25, 35], constant folding [24], variable folding [66], code motion [24, 25], dead code elimination [55], register allocation [6], etc. We refer readers to [54] for a survey on classic data flow analysis algorithms.

More recent works have used dynamic data flow analysis to overcome the aliasing challenges specifically in object oriented programming languages [27, 26]. Aliasing is a well-known challenge for static techniques where the equivalence of two or more

[1] As an example, an access to an array element must be considered as an access to any element.

pointer variables is in question. An static analysis either has to skip such challenging cases, or over-approximate by conservatively considering any potential equivalence. The information available at concrete execution helps to determine aliasing relationships and as a result improve the precision of data flow analysis. It has been shown that dynamic DFA can find data flow relations in programming languages like Java which were being missed by a minimal static DFA or were over-approximated by static alias analysis [26].

In the context of security and privacy, a more specialized data flow analysis is used named taint analysis. In taint analysis some data (for example untrusted input from the user) are assigned a special taint property. Throughout the taint analysis along with maintaining def-use relations, the taint property is propagated to any variable influenced by an already tainted variable. This way, taint analysis can help with identifying information leakage or memory corruption bugs [93, 53, 104, 91, 92].

In Chapter 3 we enhance a dynamic taint tracking tool to construct a data flow graph and track the propagation of secret data. In Chapter 4 we developed a static taint analysis to track the propagation of memory allocations for the purpose of vulnerability detection.

2.2 Symbolic Execution

Symbolic execution is another classic program analysis technique proposed in the mid 1970s which gained a lot of applications and popularity in last decade specifically in security. In contrast to concrete execution where the program is run on a specific set of inputs and a single control flow path is traversed, symbolic execution explores multiple control flow paths by assuming symbolic inputs [9, 48, 59]. As a result it can yield strong guarantees for property checking, i.e. no division by zero or no NULL dereference happen throughout the code.

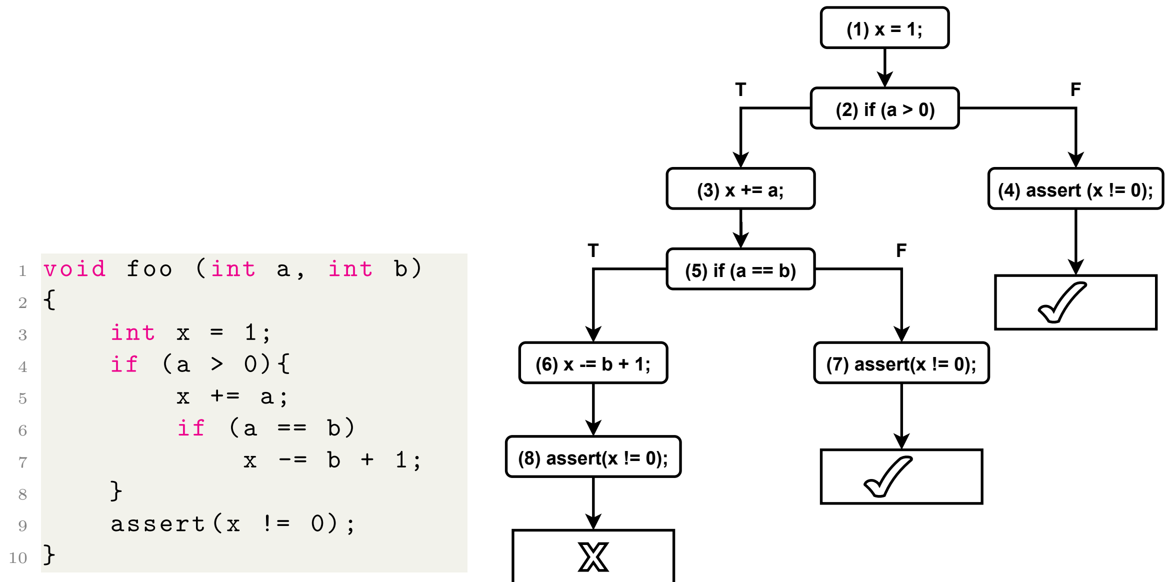


Figure 2.1: Example of Symbolic Execution

Symbolic execution allows the program inputs take symbolic values and explores all feasible control flow paths. The symbolic execution engine maintains a state consisting of path condition, symbolic store, and the next statement to be executed. Path conditions is a boolean formula describing conditions satisfied up to the current execution point. The symbolic store is map from the variables to either concrete values or symbolic expressions. Branch statements update the path condition and assignment statements update symbolic store. At each point an SMT solver can check the feasibility of path conditions and prune any infeasible state.

Figure 2.1 shows a simple function and its associated symbolic execution tree. Each node shows the state number and the statement about to execute. We aim to find the values of arguments that make the assertion at line 9 fail. To represent a symbolic value we use α. Initially the function `foo` arguments are marked symbolic, and path condition is True. So for state (1) the symbolic store is of the form $\{a \mapsto \alpha_a, b \mapsto \alpha_b\}$. In state (2) the symbolic store is $\{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1\}$. As the condition at line 4 is executed the symbolic engine forks to two states of (3) and (4)

and updates the path conditions. For state (3) the path condition updates to $\{\alpha_a > 0\}$ while for state (4) the path condition will be $\{\alpha_a \leq 0\}$. The arithmetic operations update the values in symbolic store map. Therefore, in state (5) the map will be updated to $\{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto \alpha_a + 1\}$. Again, at state (5) the symbolic engine forks to two states (6) and (7) and updates the path conditions appropriately. More specifically, the path condition for state (6) will be $\{\alpha_a > 0 \wedge \alpha_a = \alpha_b\}$. The path condition for state (7) will be $\{\alpha_a > 0 \wedge \alpha_a \neq \alpha_b\}$. The arithmetic operation at state (6) updates the symbolic store at state (8) to $\{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto \alpha_a + 1 - (\alpha_b + 1)\}$. After expanding all states until reaching the `assert` at line 9, we can evaluate for what input values of `a` and `b`, the assertion may fail. Checking the conditions for states (4), (7), and (8) reveals that only state (8) can make $x \neq 0$ false. In other words, the condition $\{\alpha_a > 0 \wedge \alpha_a = \alpha_b \wedge \alpha_a + 1 - (\alpha_b + 1) = 0\}$ is satisfiable.

Symbolically executing real world programs often encounters scalability issues. Language constructs like loops can increase the number of candidate states exponentially. Therefore, exhaustively exploring all possible states does not scale. Additionally, different modes of memory addressing and complex data structures, side effects from libraries and system code, and complex SMT formulas pose challenges for the symbolic execution engine. State of the art symbolic execution tools make different assumptions and simplifications to address such challenges [38, 22, 17, 14, 13]. An in depth survey on symbolic execution systems can be found in [4].

One of main limitations of classic symbolic execution is that it cannot reach many feasible paths due to the resulting complicated path constraints [15]. A practical idea to help with complex constraints and side effects of whole software stack is to combine concrete and symbolic execution. Such techniques are generally named *concolic execution* where the concrete execution of the program drives the symbolic execution [62]. These techniques were proposed for test input generation, where the engine maintains a concrete store along with symbolic store, and path constraints.

The engine starts execution by some randomly selected or user provided concrete inputs. It executes the program while updating path constraints and the two stores. At each branching, symbolic execution takes same direction as concrete execution and updates path conditions accordingly. Therefore the constraint solver is not invoked on every branch. To explore different paths, the path conditions at some selected branches can be negated and passed to SMT solver for feasibility check and new input generation. This workflow can be repeated which helps in achieving higher coverage in practice. DART [42] and SAGE [43] are two well known example of concolic execution tools. They employ different strategies in choosing which branch to negate next and generate new paths. $S2E$ takes a different approach in combining concrete and symbolic execution by proposing *selective symbolic execution* [21, 22]. It allows selecting to symbolically execute only parts of the code by carefully changing the execution mode from concrete to symbolic and vice versa. For example, assume function F calls function G and the mode change is at the call site. If the change is from concolic to symbolic, the arguments of G are made symbolic and it is explored symbolically along with the concrete execution. Once finished, the concrete results of G is returned to F, and F is continued concretely. If the change is from symbolic to concrete, the arguments to G are concretized and after finishing G concretely, the execution continues symbolically in F.

Path selection is another challenge for practical symbolic execution. That is because enumerating all feasible paths is impractical so different tools based on their design goal employ different heuristics to prioritize which path to explore next. Besides classic depth-first and breadth-first search algorithms that prioritize deep or shallow paths respectively, probabilistic path selection is used extensively. For example, KLEE [13] assigns probabilities to paths based on their length and favors less explored ones to prevent starvation. Other heuristics designed for better code coverage have been discussed in literature [14, 13, 22]. For example in [63] aims to

explore less traversed parts of global control flow graph by maintaining a frequency distribution for explored paths. In pursuit of finding memory corruption bugs, [17] gives priority to paths with symbolic memory addresses or symbolic pointers.

Symbolic backward execution [28] is a variant of symbolic execution techniques where the goal is to identify program inputs that can trigger execution of a specific point the code. Therefore, the analysis starts from the target point in the code and moves in the reverse direction of control flow. path conditions are collected as forwards symbolic execution. At any point that a path is deemed infeasible, the engine discards the path and backtracks to start a new one. A mixture of forwards and backward symbolic execution is presented in [72]. The technique is named *call-chain backward symbolic execution*, and starts from the function entry point of containing the target code. Once feasible paths are found in this function, the tool selects one of the function's callers and looks for feasible paths from the caller entry point to the target code. This process is repeated until a feasible path to target is found from the main function of the program. The paths are prioritized based on a shortest-path distance to the target in the inter-procedural control flow graph.

A practical approximation to avoid path explosion problem is reducing the code to be analyzed. Under-constrained symbolic execution [32] serves to analyze a function in isolation. To do so, the function input and any global data that may affect the function's execution is marked symbolic and then the engine starts as normal from the beginning of the function. Essentially a variable is under-constrained if the engine does not collect the constraints on its value along the path from program's entry point up to the function. Therefore, there can be paths deemed as feasible while they actually are not (it leads to false positive). That said, under-constrained symbolic execution is proved to be more scalable and successful in finding bugs in larger code [85, 10]. More specifically Sys [10] uses predefined static checkers to collect potentially buggy code locations and then employs under-constrained symbolic

execution on small code snippets to confirm the bugs. It marks the function arguments and any locally allocated variables as under-constrained, and assumes memory pointers cannot alias.

In Chapter 3 we use symbolic execution on a smaller regions of the code tandem with bit-level taint analysis to mark exact symbolic data at the start of region. In Chapter 4 we propose using a flavor of under-constrained symbolic execution to prune infeasible paths and improve the precision of vulnerability detection.

2.3 Static Vulnerability Detection

In this section we review related work to our project presented in Chapter 4. More specifically we look at static vulnerability detection techniques with a focus on ones that cover an OS kernel.

Static Memory Leak Detection. Hector [90] aims to detect resource release bugs in big code bases. It has been applied to the Linux kernel, but it is limited to finding inconsistencies within a single function. It first annotates any resource acquire/release operation at the function level, and then checks that each acquired resource is paired with a release on any error handling path. Error-handling paths are identified using the return value check. Any pointer-returning function is considered a resource acquisition, and the associated release function is the last one taking the acquired resource as an argument. Such broad definitions of resource acquisition and release are mainly because Hector is designed as a general resource release bug detector, not specifically for kernel memory leak detection. For example, based on our study, there are 22675 pointer-returning functions in the Linux kernel. Hector warns on situations where a resource is released on some paths through a function and not on others. so it does not detect situations in which all the paths through a function fail to release a resource. It also checks whether a resource is returned from

a function, but not if it escapes via a pointer.

SATURN [102] tries to detect memory leaks via static path sensitive pointer analysis. It reduces the memory leak detection problem to a Boolean satisfiability problem, and then uses a SAT-solver to identify potential bugs. Other static memory leak detection techniques have been proposed [96, 86, 39, 52, 78, 20], which to the best of our knowledge were not scalable to OS kernel.

Deviation-based Analysis. Our use of mining in Chapter 4 is similar to Engler et al.'s [31] proposal to infer bugs by looking for deviations from commonly observed behavior. More specifically, they looked for NULL-pointer inconsistency through the OS kernel and were able to detect multiple bugs.

Pattern mining has been employed in previous research for the purpose of program analysis. In [65] the authors used mining on code revision history to find paired functions. Found pairs are then checked at the run-time to find violations. The approach is not flow sensitive, and relies on user input to enhance pattern matching. The authors of [99, 45] applied sequential pattern mining for specification mining by focusing on Java exception-handling code. Weimer and Necula [99] proposed specification mining by focusing on error-handling code. They applied sequential pattern mining to user-level programs written in Java. PR-Miner [64] uses frequent item-set mining to infer programming rules without using user-defined templates. It does not consider the sequence of operations. MUVI [70] uses a similar approach to detect concurrency bugs. CHRONICLER [84] integrates sequential pattern mining with a path-sensitive data flow analysis to identify precedence rules for a function call. To the best of our knowledge none of these works were applicable to a large system like an OS kernel.

Error-handling based Detection. Using error-handling code to detect bugs was employed previously is many works. LRSan [97] and Crix [67] find classes of missing-check bugs in the Linux kernel via employing error-handling code to identify

critical variables. Other techniques analyzing error-handling code within the Linux kernel include [46, 88, 3] where the error propagation is evaluated to detect potential bugs. These techniques rely on explicit `errno` returning to identify error-handling code. Based on our study, non-explicit error-handling cases are common, and missing those, causes false negative. In our proposed system, such error-handling cases are covered via critical check identification as described in Section 4.4.2.

Kernel Vulnerability Analysis. Because of the inherent complexity of the OS kernel, analyzing the whole kernel is challenging. Therefore, first compiling the whole kernel into LLVM IR became a practical approach. This facilitates the analysis by employing LLVM passes. K-Miner [40] performs inter-procedural analysis by extracting execution paths starting from system-calls to detect memory corruption vulnerabilities. Without the ownership reasoning mechanism and the identification of specialized allocators/deallocators, K-Miner has to analyze complicated data flows globally, which is very hard. As a result it only focuses on the paths starting from system calls. Dr. Checker [73] finds vulnerabilities in the Linux kernel drivers via static data flow analysis. KINT [98] and UniSan [68] employ taint analysis to find integer overflow and information leakage, respectively. SLAKE [19] provides an automated method to exploit vulnerabilities in the Linux kernel by extending LLVM for its static analysis, in tandem with the fuzzer Syzkaller [44]. DCUAF [2] proposes a static analysis approach to detect use-after-free bugs in the Linux device drivers.

Chapter 3

Quantitative Information Flow Analysis

Information-flow analysis aims to track information propagation throughout a program in order to enforce limitations or detect property violations. A common example is a program working on some secret input and generating publicly available output. A security property can be that the program should not leak the secret input. From data-flow perspective, it is almost impossible to avoid flows from the secret input to the public output, but the security property can be like identifying unintended flows from the secret input to the public output. A taint-based flow analysis suffices for such scenario, where values in the program become tainted if they may contain any amount of the secret data. The taint propagates throughout the program via a simple rule: the result of an operation becomes tainted if any of the operands are tainted. For example, copying a tainted value, transfers taint to the copy as well. The problem with taint analysis is its binary nature: a value either is tainted or not. Additionally, most implementations of taint analysis do not account for implicit flows. In other words they miss to track indirect data relationship throughout the program like previous control flow transfers leading the execution to a specific point.

Quantitative information-flow analysis, aims to answer the question of how many bits of information from the secret input is revealed in the public output. Therefore,

in this scenario a taint-based approach in not sufficient because the analysis needs to measure the revealed bits and detect any violation if the amount is beyond a threshold determined by the security policy. In this chapter, we discuss a quantitative information flow analysis technique which models the data-flow as a network capacity problem. The maximum possible information-flow from the network source (secret input) to the network sink (public output), is measured by a max-flow algorithm. Then we propose a symbolic execution enhancement that improves the precision of the analysis without sacrificing the soundness.

3.1 Information Flow as Network Capacity

As mentioned, the quantitative information-flow analysis measures the bits that a program actually reveals from its input. The technique that we are about to describe is first proposed by McCamant and Ernst [75]. The tool named FlowCheck, guarantees soundness by always overestimating the actual information flow. It employs bit-level taint tracking, and also accounts for implicit data flows. As mentioned before, the taint tracking can identify presence of data flow but cannot precisely measure it. For example, if a secret data of 8-bits is copied into variable x, the taint propagates to all 8-bits, but the total amount of secret information is still unchanged. FlowCheck models the information flow as a network capacity problem. All possible information-flow streams are explored to form a network of volumed channels, and the secret information is modeled as incompressible fluid that might flow in those channels. Then the maximum secret information that the program execution might reveal is associated with the maximum rate of the fluid that can flow through the network. Such maximum rate can be measured by the application of a max-flow algorithm to the network. Based on this analogy, the weight of a minimum-cut represents the maximum possible capacity for such network. A minimum-cut is a minimal set of

the network edges that if removed, the network source disconnects from the network sink. The weight associated with each edge represents how many bits of information the edge carries.

3.1.1 Constructing the Network of Flow

Here we describe how to build a network of flow representing possible information streams corresponding to an execution of a program [75]. The flow graph consists of nodes representing operations and edges representing values. Each edge has a capacity equal to the bits of data it can hold. Edges are directed and for each node, the ingress edges are the operands of the node's operation and there is a single egress edge for each node associated with the result of the operation. Load and store operations are broken into bytes. The generated flow graph is directed and acyclic with edges pointing from older nodes to newer ones in terms of the execution sequence. The secret inputs are represented by a source node and all public outputs are represented by a sink node.

In an execution, along with direct data flows corresponding to the explicit operations on the data, there exist *implicit* data flows as well. Implicit data flows are associated to operations like branch, pointer, and array that are affected by secret data. Such operations affect control flow or the operands of other operations based on the secret data value. For example a two-way conditional branch on a secret value, can reveal 1 bit of information about the secret value. Therefore, a sound flow network should account for implicit flows as well. FlowCheck assumes an execution as an *enclosed* computation with determined inputs and outputs. Then for each identified implicit flow, an edge connects the implicit operation to the outputs. By default, the program outputs are the receiver of the implicit edges, but an special annotation can be used at the source level to define more precise computation units with determined outputs. Such annotation (named *enclosure region*) helps with the precision of the

calculated information flow at the end. The capacity associated with the implicit flow edges is determined by the number of different executions for that operation. For example for a two-way branch on the secret data, the capacity will be one bit, while for a pointer operation like indirect memory access or jump, the capacity will be equal to the number of secret bits in the pointer.

Each edge in the constructed flow network is weighted. The weight or capacity of an edge represents the maximum amount of information carried by that edge. The capacity for direct flow edges are computed using bit-level dynamic taint analysis. For each memory location and registers in the program execution context, a shadow bit vector is maintained to mark which bits are secret. These marked bits are calculated based on bit-level taint rules for basic operations. Then, the number of secret bits determines the capacity of the associated edge in the network.

3.1.2 Calculating the Maximum Flow

Once FlowCheck constructs the flow network, calculating the maximum possible information flow from the secret inputs to the outputs is straightforward. Based on the max-flow min-cut theorem, the maximum capacity transferable from the network source to the sink is the capacity of a minimum cut. A minimum cut is a set of edges that if removed, disconnect the network source and sink. The weight of the minimum cut is the sum of the capacity of edges in such set. This value represents a conservative estimation of information leaking from the secret input to the public output.

3.2 Symbolic Execution for Better Precision

Even though FlowCheck accounts for implicit flows and employs bit-level taint analysis, in some scenarios it still unnecessarily overestimates the information flow. For

```
1 if (i < 16)
2     v = base + i;
3 else
4     v = base;
```

Figure 3.1: An example of overestimation by FlowCheck

example, in the code snippet in Figure 3.1, assuming the type of the variables to be unsigned 32-bit integer, the total information flow form the secret variable `i` to the output `v` is 4 bits. That is because the check at line 1, limits the influence of the secret value `i`: for any possible values of `i`, the value of `v` will be in the range of $[base, base{+}15]$, which means quantitative influence is 4 bits. This is while FlowCheck reports 32 bits as dynamic taint analysis marks `v` as tainted when the check at line 1 passes.

We can do better by calculating the actual control that an input has over the output. Newsome et al. [77] proposed the notion of *influence* to measure such control to detect integrity policy violations. In a deterministic computation, the influence of the computation input over the output is $\log_2$ of number of possible output values. We can see that the influence of `i` over `v` in Figure 3.1 is 4 bits.

To measure the influence of a computation we can use symbolic execution to formulate the computation's output as a function on its inputs. In this way, the computation execution paths are expressed as SMT formulas over bit-vectors. Generally such formulas are passed to a SMT solver, where the solver checks for satisfiability. But in our case we need to determine the number of possible output values, so we use *model counting*. Model counting techniques are used for enumerating the number of satisfying assignment for a given formula. This makes it suitable for quantitative information analysis, and more specifically in our case: influence measurement. This way, for a given execution region, we measure the input influence by counting the number of distinct output values by varying the input bits.

To improve the precision of dynamic taint tracking, we employ Fuzzball [74, 38] as our symbolic execution engine, and SearchMC [58] for model counting. FuzzBALL is a binary symbolic execution tool which uses Vine library from the BitBlaze framework [95] to lift the binary and instrument it. SearchMC is an approximate model counting tool that uses statistical estimation to compute a model count of a SMT formula. As it measures influence on a statistical estimate, it may return a fractional result for a formula, which we round it up, and then use it in our information flow analysis. This round-up is most cases compensates for the randomness of SearchMC. For a detailed analysis of confidence in SearchMC result we refer the reader to [58].

For a given execution region, the symbolic execution engine extracts SMT formulas and passes them to the model counter and then, we use the measured influence to refine overall information flow. The main idea is finding regions of execution that influence measurement can give us more precise estimate on information flow and then use this estimate to replace the one of taint analysis. We need to select the regions in a way that symbolic execution can compute in a reasonable time. This method is still a sound information flow analysis as the symbolic execution step explores all execution paths of the region, therefore the influence measurement is still an overestimate of the information flow (within a confidence interval as an argument to the model counter), but a tighter one.

Once the symbolic execution calculates the influence for a given region, we update the flow network with the influence measured for the region. We call this process *surgery*. The surgery replaces the sub-graph representing the region with an edge from the region's inputs to the region's output. The replacement edge's weight is the measured influence via model counting. Re-evaluating minimum cut, will reflect the influence measurement in overall information flow analysis of the program. For example, the sub-graph representing the code in Figure 3.1 will be replaced with an edge from the `i` to `v` with a weight of 4. Re-evaluating minimum cut reflects this new

measurement in the overall flow estimate.

3.2.1 Internals of FCFB

We named our hybrid quantitative information analysis tool FCFB as a concatenation of the abbreviated names of two existing tools we are using: FlowCheck and FuzzBALL.

As preprocessing step, given a subject binary for analysis, we statically extract the call graph, function boundaries, and control-flow graph of each function. Additionally, for each basic block, we identify pre-dominator and post-dominator. For the preprocessing steps we developed an static tool using DyninstAPI [11]. DyninstAPI provides interfaces for platform independent binary analysis and transformation.

We then run the subject binary under FlowCheck. This run creates the initial data flow graph based on the notion of network capacity and gives us the min-cut. We model an execution point as a pair of an instruction address and instruction count. The second element of the pair is a count of executions for the first element. This way we can differentiate between multiple execution of a single instruction (like in a loop). This pair helps to pinpoint same point of execution which is important to transfer between FlowCheck (data flow analysis) and FuzzBALL (symbolic execution). Therefore the identified min-cut location is a set of pairs of the form `(mc_addr, mc_count)`. Using this set we extract an snapshot of whole execution at the point of min-cut. Meaning that we create an snapshot of instruction counts at the point of min-cut. These information is used when passing a region to FuzzBALL for influence measurement.

The initial region that can be considered for influence measurement using symbolic execution, is the smallest region covering all cut edges. For a simple case where the min-cut consists of a single edge, the basic block containing `mc_addr` will be the starting region. Meaning that the region of code to be passed to FuzzBALL will start

at the first instruction of the basic block and end at the last instruction of the basic block. Using the instruction count snapshot maintained at `(mc_addr, mc_count)`, we can identify the instruction count for these start and end addresses. Therefore, the influence measurement starts at `(start_addr, start_count)` and finishes at `(end_addr, end_count)`.

Next, we have to identify the inputs to the region and outputs from the region. The input to the region are the tainted data at `(start_addr, start_count)`. The output from the region are the newly tainted data at `(end_addr, end_count)`. We implemented new features in FlowCheck to extract tainted data at any execution point. Therefore, we once extract tainted data at `(start_addr, start_count)` calling it set T1, and once extract tainted data at `(end_addr, end_count)` calling it set T2. T1 is the input data to the region, and the output from the region is $T2 \setminus T1$.

Now that we have the execution region along with the input to and output from the region, we go ahead and issue an influence measurement query to the FuzzBALL. This query starts at `(start_addr, start_count)` and ends at `(end_addr, end_count)`, while marking memory regions or registers in T1 as symbolic. It measures the influence of those symbolic inputs on the data in $T2 \setminus T1$.

Once FuzzBALL finishes, FCFB extracts the measured influence and performs surgery. As described before, surgery updates the network capacity graph by replacing the subgraph associated to the execution region with an edge from the region's inputs to the outputs. The surgery edge is labeled with a weight of measured influence. Then we feed this updated network capacity graph to the max-flow algorithm. The newly calculated max-flow reflects the effect of influence measurement for the chosen execution region.

3.2.2 Region Exploration

FCFB uses min-cut as a hint to start off from an interesting execution point. The min-cut is interesting as it is somehow the bottleneck of the network capacity graph and it pertains the critical data dependency between program input and the output. In Section 3.2.1 we started from the basic block containing the min-cut. It will be more fruitful if we can expand the region and cover more execution area. But it is not practical to expand the region in a way that covers the whole program. That is because influence measurement is very expensive inherently because of the underlying symbolic execution and model counting. We can expand the region by multiple step sizes. For example, the start address can be set to the previous basic block in the CFG. Same for the end address, which can be set to the next basic block in the CFG. Because a basic block may have multiple immediate predecessors and successors, using pre-dominator and post-dominator is a more consistent approach. Additionally, combinations of start and end exploration can be employed.

As described in Section 3.2.1, the initial code region passed to FuzzBALL is derived based on the min-cut. The code region should respect the invariant that the start point precedes the end point, otherwise the symbolic execution fails. When expanding current region, we have to guarantee that this invariant holds for all the candidate regions. One possible approach is incorporating pre- and post-dominator relationship for the current function's CFG. This becomes challenging when the region expands beyond function calls. Therefore, in pre-processing phase we extract the chronological ordering of the execution addresses along with their execution count. We do this by instrumenting FlowCheck to dump current instruction address and count when generating the initial data flow graph. Recall that we model an execution point as a pair of address and count. So, the chronological execution dump gives us all execution points of the subject binary on the given inputs. The code region used for symbolic execution can be mapped to the execution points in the chronological dump.

$(i-ss, j+ss)$	$(i, j+ss)$	$(i+ss, j+ss)$
$(i-ss, j)$	(i, j)	$(i+ss, j)$
$(i-ss, j-ss)$	$(i, j-ss)$	$(i+ss, j-ss)$

Table 3.1: Neighbor regions of (i, j) with a step size of ss

Therefore, the invariant check is reduced to comparing the indexes of start and end execution points in the dump. As long as the start index is less than the end index, the invariant holds.

A given region is represented as a pair of `(start_index, end_index)` corresponding to their indexes in the chronological execution dump. Assuming an step size of `ss`, we can generated a set of 8 neighbors for a given region, as shown in Table 3.1.

Therefore, our exploration space can be represented by the upper triangle of the matrix of start and end indexes. This means we have a discrete search space. A region passed to FuzzBALL may either produce an influence measurement or fail to do so (e.g due to timeout). As a result, we cannot make any assumption on monotonicity of the calculations.

These all suggest that we have a discrete non-convex optimization problem to find the best region to minimize the overall data flow analysis. Greedy approaches like gradient descent [81, 89] that only select a neighbor with better result for the current state may end up in local optima. Instead, we adapt an approach based on Simulated Annealing [60, 29] to search the regions. Simulated Annealing is an iterative probabilistic local search algorithm for finding a global minimum. The naming comes from an analogy in thermodynamics were a crystalline solid is heated first and then left to cool down slowly to reach its regular crystal lattice. As the solid cools down the crystals movement decrease in a way that eventually it ends up with no crystal defect. The main advantage of Simulated annealing over techniques like gradient descent is its ability to escape local minimums by moving uphill with a probability [47]. In a discrete optimization problem, each iteration of Simulated Annealing consists

Algorithm 1: Simulated Annealing Algorithm for Region Exploration

```
Initialization: Specify (Ω, f, N), an initial state w ∈ Ω, a temperature
  parameter t_k, k = 0, 1, ...
while Stopping criterion not met do
    Generate a neighbor w′ ∈ N(w)
    Calculate Δ_{w,w′} ← f(w′) − f(w)
    if Δ_{w,w′} ≤ 0 then
        w ← w′
    else
        Generate a random number u ∼ U(0, 1)
        if u ≤ exp(−Δ_{w,w′}/t_k) then
            w ← w′
        end
    end
    k ← k + 1
end
```

evaluating the objective function of a neighbor and comparing it against the current state. If the neighbor generates an improved solution, it is taken unconditionally. A non-improving neighbor can also be chosen based on a probability function. Such non-improving steps are performed to escape local optimum solutions. The probability function to accept a non-improving solution depends on the current temperature. As the algorithm iterates, the temperature decreases and as a result non-improving solutions are chosen less frequently. Multiple research [29, 50, 47, 34, 61] have covered the theoretical aspects and applicability of Simulated Annealing which are out of the scope of this thesis.

Algorithm 1 depicts the adapted Simulated Annealing from [50] that we are using for our region exploration problem to find optimal data flow estimate. To describe the algorithm, we provide the following definitions:

- **Set of all possible states** Ω**:** Each state is in the form of `(start_index, end_index)` corresponding to the indexes in execution dump.

- **Objective function f:** Let $f : \Omega \to \mathbb{R}$, the goal is finding a global minimum $w^* \in \Omega$, such that $f(w) \geq f(w^*)$ for all $w \in \Omega$. The objective function must be bounded to ensure such global minimum exists. For each state, we run FuzzBALL on it and produce a final flow estimate. This estimate is bounded in the range of zero and original FlowCheck estimate.

- **Neighborhood function N:** The neighboring function takes current state w as input and returns one of its possible neighbors randomly.

The initial state w is generated based on the min-cut. The temperature parameter t_k decreases as k increases. We chose a linear decrease of temperature at each iteration. At each state, we take a neighbor uniformly from the set of all possible neighbors for w and use it for influence measurement using FuzzBALL. The calculated influence is then used to estimate a new flow value. If this estimate is improving the overall measurement, the the neighbor is selected for the next iteration. Otherwise, it is selected with probability of $exp(-\Delta_{w,w'}/t_k)$. As t_k decreases, such probability decreases, too. Therefore, non-improving neighbors are selected less frequently. For the stopping criterion we use a timeout threshold of 2 hours.

3.3 Case Studies

In this section, we evaluate our proposed hybrid method by applying it to three use cases: character counting, image transformation, and checksum calculation.

3.3.1 Count Vowels Binary

Assume a scenario where you have a program in binary format which returns the ratio of vowel characters in an input text file. If the input text file contains sensitive information, we do not want the amount of information leaked by the program surpass

```
1  int num_letters = 0;
2  int num_vowels = 0;
3  int bar_len;
4  ...
5  while ((num_read = fread(buf, 1, 4096, fh))) {
6      count_vowels(buf, num_read);
7      total_read += num_read;
8  }
9
10 if (num_letters == 0) {
11     bar_len = 0;
12 } else if (num_vowels > num_letters){
13     bar_len = 70;
14 }
15 else {
16     bar_len = (70 * num_vowels) / num_letters;
17 }
18
19 for (i = 0; i < bar_len; i++) {
20     print_buf[i] = '#';
21 }
22 ...
```

Figure 3.2: Main Part of Count Vowel Code

a security threshold. For this reason we apply our quantitative information flow analysis on the Count Vowels Binary (CVB) to measure the amount of information leaked by the CVB's output.

Figure 3.2 shows the main part of CVB source code. The `while` loop at line 5 reads the contents of the input file and calls `count_vowels()` on the read buffer. This function checks each character in the buffer and counts the number of observed vowel characters and total characters by updating two global variables: `num_vowels` and `num_letters`. CVB asserts that these two variables are always non-negative. The code then calculates a `bar_len` using the two global variables, to represent the ratio of vowel characters in the input text (lines 10 through 17). This `bar_len` variable determines the number of `#` characters to be placed in the `print_buf` which later will be displayed to the user.

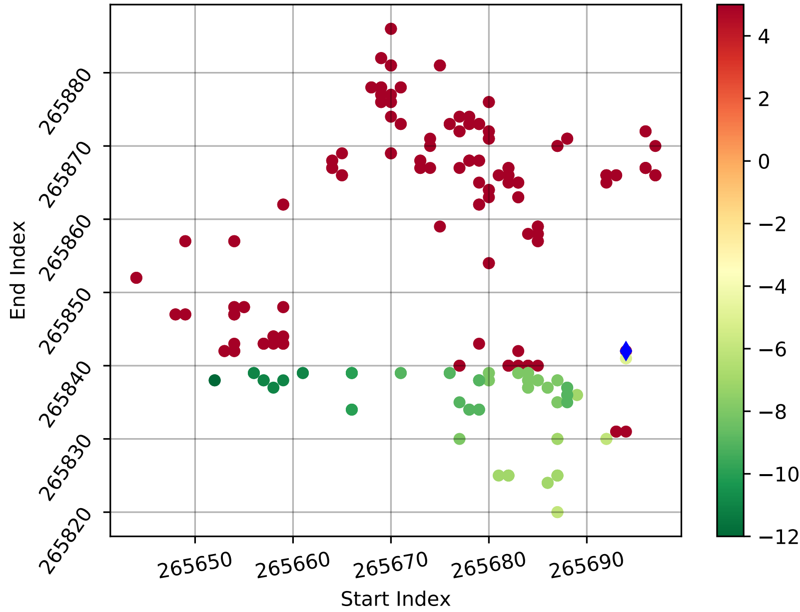


Figure 3.3: FCFB region exploration for `CVB`

The public output of CVB is the `print_buf` array. Which its number of elements is bounded by `bar_len` variable. `bar_len`, in turn is being calculated based on two variables `num_vowels` and `num_letters` which are based on the secret input file contents. Assuming that the `bar_len` variable can fit in a 32-bit signed integer, we can say that the maximum information leak cannot be more than 32 bits. A closer look into the calculation of the `bar_len` can give us a tighter bound. Lines 10 through 17 confirms that the value of `bar_len` will be in the range of $[0, 70]$. In other terms, the number of `#` printed is bounded in the range $[0, 70]$ which is about 7 bits of information. In the code presented in Figure 3.2 we left out the details on how to enclose implicit data flows. In the actual code to be run under FlowCheck, we use an enclosure region starting at line 9 and ending at line 18 with the specific output of `bar_len` to account for the implicit data flow of the `if-else` clause on `bar_len`. More details on enclosure regions is provided in [75].

```
uint8_t *fileContent;
uint16_t *alphaBuf;
int i, percent = 90;
...
for (i = imageBodyIndex, j = 0; i < fileSize; i += 3, j++){
    r = fileContent[i];
    g = fileContent[i+1];
    b = fileContent[i+2];
    gray = (299*r + 587*g + 114*b)/1000;
    alphaBuf[i] = 257 * ((r*(100-percent) + gray*percent)/100);
    alphaBuf[i+1] = 257 * ((g*(100-percent) + gray*percent)/100);
    alphaBuf[i+2] = 257 * ((b*(100-percent) + gray*percent)/100);
}
...
```

Figure 3.4: Main Part of De-saturate Transform Code

Running CVB on its source code under FlowCheck reports 24 bits of information leak. This is while our hybrid method can identify the tighter range of values for `bar_len` and reports 7 bits of information leak.

Figure 3.3 shows the progress of FCFB region exploration for our experiment. The X axis is start index in the execution dump, and Y axis is the end index. Each dot on in the graph represents a region, which is used for influence measurement. The color-bar to the right, shows the the improvement of the result; the greener, the more improvement in terms of information leak estimate. The exploration started from the diamond, and eventually reached the best answer of 7. The best region for the CVB was a region covering lines 10 through 17 in the code of Figure 3.2. For this region, the secret inputs are `num_letters` and `num_vowels` and the output is `bar_len`. The symbolic execution measure an influence of 6.4977 bits from the region inputs to the output and the final max-flow measurement reports 7 bits of information leak.

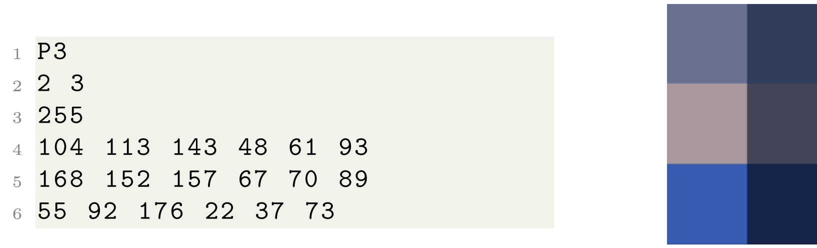
3.3.2 De-saturate Image Transformation Binary

As another case study, we apply our hybrid method on an image transformation binary. Here, the code reads in a RGB image in PPM format, and for each pixel de-saturates it by eliminating hue and saturation information while retaining the luminance. Figure 3.4 shows the main part of this transformation.

The `for` loop at line 5 iterates over each pixel and calculates a `gray` value, then the output pixel is derived as a weighted sum of the input R/G/B with the calculated `gray` value. The gray value calculation at line 9 is based on the formula used by `rgb2gray` function in MATLAB [87]. In our experiment we used 90 for the value of `percent`. To calculate the actual information leak, we need to look into the internals of the PPM format and the way the output pixel values are calculated.

Portable PixMap (PPM) image format is encoded in human-readable text [83]. Figure 3.5 shows the contents of the input image we used for our case study. Lines 1-3 are the image header that provides an overview of the rest of the image. The header consists of four entries in the same layout shown in lines 1-3. The first line determines the image format (P3). It says this file is a full-color, ASCII text encoded PPM image. Line 2, determines the number of columns and rows in the image. So, the example is a 2 pixel by 3 pixel image. Line 3 is called maximum color value, which determines the range of values used for color intensity. Commonly, 255 is used as maximum color value and it means that for each pixel, the value of red, green, or blue can range from 0 up to 255. Lines 4 through 6 contain the image body. Each pixel is represented with triple value determining the intensity of red, green, and blue. So, a pixel with values of 0 0 0 will be black, and with values of 255 255 255 will be white.

Running de-saturate code with the input image, under FlowCheck gives an upper bound of 168 bits of information leak. This is while the information leak reported by our hybrid approach is 152. To evaluate the actual information flow in the de-



```
P3
2 3
255
104 113 143 48 61 93
168 152 157 67 70 89
55 92 176 22 37 73
```

Figure 3.5: Input PPM Image

saturate binary, we have to account that 24 bits of information associated with the header of the ppm file is copied to the output directly. The amount of information transferred to the output pixel varies depending on the `percent` value. For example, for the case of $percent = 0$ all of $3 \times 8 = 24$ bits are copied to the output. For our case, where $percent = 90$ we calculated the influence empirically: for each value of red/green/blue we have 256 different values therefore we could apply the computation shown in Figure 3.4 with all possible values for r, g, and b as input to generate the new red/green/blue value. We then count the number of set bits in a matrix of 3×256. This way we derived the influence of input pixel on the output pixel to be: $\log_2 179836 = 17.456322$. As a result, the actual information leak will be $24 + 6 * 17.456322 = 128.737932$. Figure 3.6 shows the progress of FCFB region exploration for our experiment. The exploration started from the blue diamond, and eventually reached the best answer of 152. The best region found by FCFB for this binary were equivalent to the code of line 9 through 12 in Figure 3.4 where it calculated an influence of 8.5329 bits from the region secret inputs to the output.

3.3.3 cksum Binary (coreutils)

`cksum` is a program in coreutils that takes an input file and prints the CRC-32 checksum of the file contents along with its byte counts and the file name. Cyclic Redundancy Codes (CRC), is one of the most common error-detection techniques used

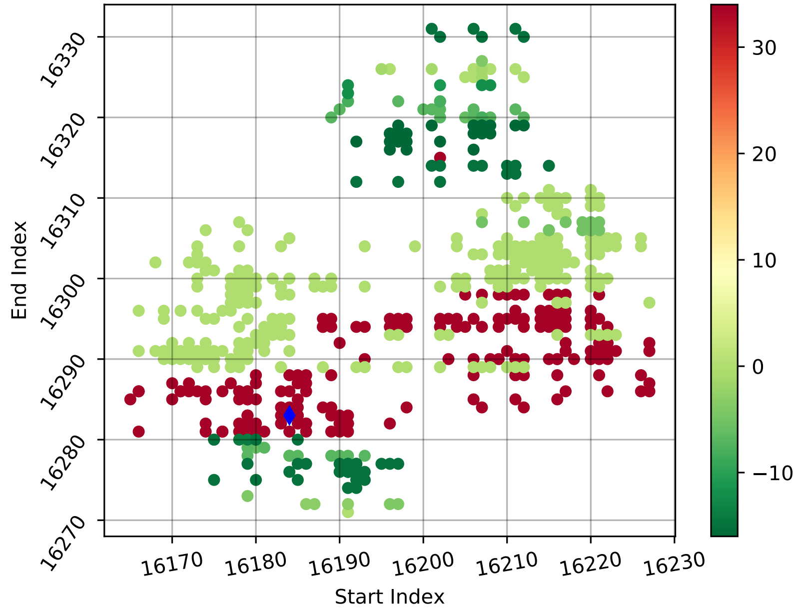


Figure 3.6: FCFB region exploration for `de-saturate` binary

both in software and hardware. The basic idea behind CRC algorithm is to treat a message as a binary number and to divide it by another fixed binary number. The division's remainder is the checksum that can be used for integrity check [100]. Detailed description of CRC implementations can be found in [82, 100]. It is now well known that CRC is not secure to use in adversarial contexts [8, 37].

In general hash functions and in some cases specifically CRC functions are presented as examples of functionalities that are hard to symbolically execute [12, 41, 51]. Sharma et al. [94] have shown that CRC calculation may not impede symbolic execution if some clever optimizations are employed. For our experiments in this section, we have used theory-of-arrays-table optimization to create formulas reflecting contents of the underlying lookup table. That is because as shown in [94], this optimization works the best for CRC-32 with shorter symbolic inputs.

`cksum` uses the table-driven implementation of CRC. Such implementation is faster

```
1  while ((bytes_read = fread (buf, 1, BUFLEN, fp)) > 0)
2  {
3    unsigned char *cp = buf;
4
5    length += bytes_read;
6
7    while (bytes_read--)
8      crc = (crc << 8) ^ crctab[((crc >> 24) ^ *cp++) & 0xFF];
9  }
10
11 crc = ~crc & 0xFFFFFFFF;
```

Figure 3.7: Main Part of CRC-32 computation of `cksum`

as it uses a lookup table to retrieve pre-computed values for a given byte. Figure 3.7 shows the main part of the underlying code used in `cksum` for CRC-32 calculation.

In each iteration of the while loop at line 1, the input is read into the buffer. In the while loop at line 7, for each byte in the buffer, based on the content of the buffer, a pre-computed value is loaded from the table `crctab`, and XOR'd with the running CRC value. At the end, the bit sequence is complemented for the final CRC. A detailed explanation of this table-driven implementation can be found in [100].

We ran the `cksum` under FlowCheck and FCFB using a test input file of size 1983 bytes. The FlowCheck reported 73 bits of information leak, while FCFB reported 59 bits of information leak. To evaluate the actual information flow in `cksum` binary, we have to notice that the bit masking at line 11 of the code in Figure 3.7 ensures that the `crc` variable cannot carry more than 32 bits of the secret information from the input file. `cksum` also prints the byte counts of the input file. The amount of information leak as a result of byte length printing cannot be more than 21 bits. That is because each digit printing leaks $\log_2 10 = 3.3219$ and our test input has 4 digits length size. Additionally, the string format of `length`[1] variable leaks at most 5 bits. Therefore, the overall information leak of `cksum` binary for our test input

[1] `length` is of type `uintmax_t` which is 64-bit long on our test system.

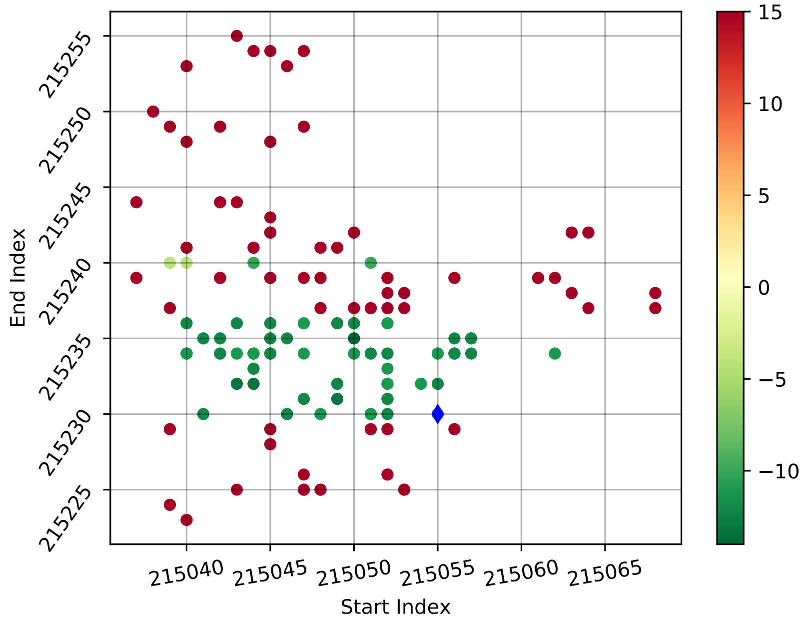


Figure 3.8: FCFB region exploration for `cksum` binary

is $32 + 4 * 3.3219 + 5 = 50.2876$. Figure 3.8 shows the progress of FCFB region exploration for our experiment. The exploration starting from the blue diamond, eventually reached the best answer of 59. The best region found by FCFB corresponds to the code lines 7 and 8 in Figure 3.7, where it calculated an influence of 20.9917 bits from the secret file contents to the `crc` variable.

Chapter 4

Static Vulnerability Detection

In this chapter we present a static vulnerability detection system that incorporates multiple data-flow-based analyses to detect resource leak bugs in operating system code. Our proposed system first identifies memory allocation functions, then goes for identifying associated deallocation functions. This pairing is important as the memory allocated through an allocating function should be released via an appropriate deallocation function. Then, our system incorporates an ownership reasoning technique to determine where in the code the allocated memory is supposed to be released. Missing a release function in such locations indicates a resource leaking bug. We also enhance the vulnerability detection with under-constrained symbolic execution to distinct false positives from true vulnerabilities.

Portions of this chapter draw on work previously published in the proceedings of the Network and Distributed System Security Symposium (NDSS) 2021 [30].

4.1 A Study of Kernel Memory Allocation

In this section, we take a look at the dynamic memory allocation mechanism in a kernel (using the Linux kernel for concreteness).

Types of memory allocation. Dynamic memory allocation in the kernel is not as

straightforward as in user-space. The complication comes mainly from the fact that the kernel has much less physical memory available—generally kernel memory is not pageable, and often the allocation is required to be a physically continuous memory region and sometimes in specific address ranges. The kernel allocation sometimes needs to be atomic i.e. it should not sleep. Such delicacies make any mistake in kernel allocation have a higher impact on the whole system's stability.

OS kernels typically allocate only a relatively small stack per thread.[1] This limitation requires kernel developers to avoid allocating large structures on the stack but instead, to perform more heap-based allocation. `kmalloc()` is the general-purpose allocation interface in the kernel. It takes two arguments: `size` and `flags`. Like user-space `malloc()`, `size` specifies the allocation size in bytes. The `flags` argument controls the behavior of the allocation. On success `kmalloc()` returns a pointer to the memory of `size` bytes, while in case of failure a NULL pointer is returned. The flags parameter in `kmalloc()` tells the kernel how or where to perform the allocation.

As an example, the flag `GFP_ATOMIC` instructs the memory allocator never to block, i.e., it cannot go to sleep to free up the required memory. This is appropriate when the code holds a lock, or in interrupt handlers. On the other hand, the flag, `GFP_KERNEL`, indicates the normal kernel allocation which may go to sleep. As another example, flag `GFP_DMA` instructs the memory allocator to allocate from the physical address range which is accessible by the hardware through direct memory access. The header `<linux/gfp.h>` defines all the allocation flags.

`kmalloc()` returns physically continuous memory. Such an allocation has two main advantages over virtual memory allocations. First, it can be used by the hardware as non-CPU devices use physical addressing. Second, physically continuous memory can be allocated within a single large page and as a result, be faster from

[1] On Linux, kernel thread stacks vary by architecture, but are commonly 1, 2, or 4 pages, so 4–16 KiB.

memory translation perspective. However, allocation via `kmalloc()` has a higher chance of failure for large sizes. Therefore if there is no need for physically continuous memory, or if the allocation size is large, `vmalloc()` should be used. `vmalloc()` uses page table manipulation to create a virtually continuous memory region. It also may block when allocating, and therefore cannot be used in an interrupt handler.

To avoid memory fragmentation, especially when many identical objects should be allocated, a slab cache can be used [7]. The cache is set up via `kmem_cache_create()` and the allocation from the cache is realized via `kmem_cache_alloc()`. For example, Linux maintains separate caches for inode, dentries, and buffer heads [57].

The dynamically allocated memory should be explicitly released when it has no further usage. Otherwise, the memory is leaked and eventually, no further allocation will be possible. The allocations via `kmalloc()` should be released via `kfree()` which takes the pointer to the memory to be released and returns the memory to the kernel. Allocations via `vmalloc()` should be released by `vfree()`, and slab cache allocations via `kmem_cache_free()`.

Specialized allocation. Various kernel modules have their own specialized allocators. Such allocators are responsible to allocate and sometimes initialize a specialized structure. The memory allocation is usually realized via a more primitive allocator, and after some initialization or extra operations, a pointer is returned to the caller. Such allocations require a specialized release and are not deallocated just by `kfree()`. For example, the `netlink`[2] module uses `nlmsg_new()` to allocate a new message and uses `nlmsg_free()` to release such message. Such specialization particularly imposes challenges to the detection of memory leaks because the detection must identify allocators and the corresponding deallocators.

[2] include/net/netlink.h

4.2 Memory Leak

An operating system (OS) kernel is part of the trusted computing base (TCB) in most modern software systems. Vulnerabilities in the OS kernel allow an adversary to bypass security measures and compromise the whole system. Much of the research in the security community has focused on memory-corruption bugs in the kernel [40, 69, 49, 101, 103]. At the same time resource exhaustion vulnerabilities in the kernel have been subject to less attention, though they too can have severe consequences on system stability and availability [18, 79, 76]. Based on our study, we found out that just 17 CVEs were assigned to Linux kernel memory leak bugs. 10 of these vulnerabilities were discovered in the past two years.

Memory is a primary resource available to a kernel, and it may be exhausted by memory leak vulnerabilities. A memory leak happens when an allocated memory region is not released even though it will never be used again. A memory region is definitely leaked when all the pointers to the allocated memory go out of scope or are overwritten. Leaked memory becomes unusable until the system reboots. Kernel memory is typically never swapped out, so a kernel memory leak reduces the physical memory available for any other purpose. Due to increased paging, a memory leak can hurt performance [33] and eventually exhaust all the available memory. Kernel memory leaks happen mostly on error-handling paths mainly because such paths are less exercised during tests. If a path associated with a memory leak gets executed frequently, the triggered memory leak eventually causes a denial of service. Furthermore, many small or infrequent memory leak bugs may lead to the same situation [16, 23]. Of course, even if a memory leak occurs only very rarely in normal operation of a system, a malicious user may find a way to trigger a leak with high frequency.

For example, Figure 4.1 depicts a memory leak bug that was discovered by our tool which was assigned `CVE-2019-19062`. Here the function `crypto_report()` at line

```
/* File: crypto/crypto_user_base.c */
static int crypto_report(struct sk_buff *in_skb,
    struct nlmsghdr *in_nlh, struct nlattr **attrs)
{
  struct crypto_dump_info info;
  ...
  alg = crypto_alg_match(p, 0);
  if (!alg)
    return -ENOENT;

        err = -ENOMEM;
  /* Memory is allocated here */
        skb = nlmsg_new(NLMSG_DEFAULT_SIZE, GFP_KERNEL);
        if (!skb)
                goto drop_alg;
  ...
        info.out_skb = skb;
  ...
        err = crypto_report_alg(alg, &info);

drop_alg:
        crypto_mod_put(alg);

        if (err)
                return err; /* Memory leaks here */
  ...
}
```

Figure 4.1: A new memory leak bug (CVE-2019-19062) detected by K-MeLD: the path ending at line 25 needs to release `skb`.

13 allocates the memory via `nlmsg_new()` for a netlink message. A pointer to this message is stored in `skb`. The code at line 14 checks whether the allocation succeeded or not. If the allocation was not successful, execution returns at line 15. If the allocation was successful, the execution continues up to the line 19, where function `crypto_report_alg()` is called. The status of the call to `crypto_report_alg()` is checked at line 24. If it was successful, the `skb` is consumed in lines after 26, but if `crypto_report_alg()` fails, the function returns at line 25 without releasing `skb`, which is a memory leak.

Challenges. Two key challenges exist in static memory leak detection for an OS

kernel. (1) *Specialized functions.* Monolithic OS kernels like the Linux kernel tend to contain tens of thousands of different modules developed by numerous vendors and programmers. We consider a function as specialized if it is developed for a specific module to perform a customized flavor of generic operation (like allocation/deallocation of a network buffer). As a result, an OS kernel has a large number of specialized functions for memory allocation and deallocation. An effective detection requires identifying such specialized allocation functions and the corresponding deallocation functions. While there are only a handful of commonly used allocation functions, our tool found more than 800 specialized ones.

(2) *Complicated and lengthy data flow.* A memory leak occurs when a memory object is not released at the end of the object life-cycle. In other words, an effective detection often has to analyze a lengthy data flow—from allocation to the end of life-cycle. More importantly, the lengthy data flow can be highly complicated in an OS kernel: Pointers to an allocated memory object are also often copied between different data structures across functions. An effective detection must determine which location or function is responsible to release the memory object[3]. We say a memory pointer is an *escaping pointer* if it is passed to the caller of the current function and thus can be freed outside the scope of the current code. A *consumer function*, on the other hand is a callee of current function that takes the ownership of the memory object, therefore the current function should not try to release after returning from the consumer function. An analysis needs to accurately recognize escaping pointers and consumer functions to avoid false-positive memory leak reports.

Given the challenges for static leak detection in an OS kernel, one may wonder if a dynamic approach would yield better results. We argue that a static detection works better because there is no need to provide real or synthetic inputs to trigger

[3]Incorrectly placed releases introduce severe memory corruption bugs like use-after-free or double-free.

all potential execution paths. Moreover, for specialized drivers, the specific hardware should be available to be able to exercise the driver code. Because of such inherent code coverage shortcomings of dynamic approaches, we opt for a static leak detection approach.

Researchers have attempted to statically detect kernel memory leaks; but their techniques have important limitations. In particular, general bug finding tools like Coccinelle [80] have limited effectiveness for detecting kernel memory leaks. Coccinelle does not implement any specific bug-finding policy but allows specifying patterns to search for potentially faulty code blocks in a function's CFG. For example, to find a memory leak a general pattern could be like: any allocation function should be followed by a deallocation. Such a high-level pattern yields a high rate of false positive. Because of the cost of manual rule specification, these tools have been applied just to general purpose allocation functions that have well-known deallocation counterparts (like `kmalloc-kfree`).

In addition, previously proposed systems that detect resource release bugs [102, 90] either lack support for specialized allocations, or fail to effectively handle escaping pointers and consumer functions.

In this chapter, we introduce K-MeLD (**K**ernel **Me**mory **L**eak **D**etector), a static analysis tool, to detect kernel memory leaks. K-MeLD not only identifies specialized allocation functions and the corresponding deallocation functions, but also answers where an object is supposed to be released by handling the complicated and lengthy data flow. K-MeLD features multiple new techniques. First, K-MeLD identifies specialized allocation functions by using a usage-driven and structure-aware analysis, then uses a context-aware and path-sensitive mining technique to detect corresponding release functions. More specifically, the initial set of allocation functions are populated based on the type and uses of the return value and the way such value is derived. Then, assuming the faulty dynamic allocation managements are outliers [31],

the high-level intuition is modeling the common approach of memory releasing to effectively identify memory release operations even in specialized cases. In a big corpus of code like a kernel[4], there are many potentially long execution paths. It has been shown that error-handling paths get less of testing and coding review [3, 71]. Based on our observation from manually checking the sites of memory mismanagement bugs, error-handling codes were where we could spot the bugs. In addition, error-handling code are usually shorter than normal execution paths and are intended to restore the state of system from an error. That means we can expect to find the release functions in error-handling paths if the function is supposed to release memory. The number of correctly implemented error-handling paths is much more than erroneous ones. Therefore, modeling the common behavior of error-handling paths enables us to single out potentially buggy implementations.

Second, we develop an ownership reasoning mechanism to infer the release locations. K-MeLD first uses enhanced escape analysis to determine when the ownership of an allocated object is transferred. We observed that kernels typically follow an informal discipline of "ownership" of dynamically allocated data (related to the concept codified in languages like Rust, but without type-system support). A function that allocates a memory object will either be responsible for deallocating that object itself, or if the object is made available to the caller via a return value or a pointer, the calling function also takes the responsibility to deallocate the object. Based on this pattern, our tool can avoid false positives by determining when objects can *escape* to a calling function: we do not expect a function to deallocate an object on a path where the object escapes. In such scenarios, the calling function is responsible to appropriately manage the allocated memory object. On the other hand, the allocating function may also pass a memory object down the call-graph to a callee, thus it is important to have an inter-procedural analysis to determine how the callee treats

[4]For example the Linux kernel is over 27 millions of lines of code.

the memory object. We call a function a *consumer* if it releases the received memory object or transfers its ownership on all or some of its execution paths. This is important because once returning from the callee, the allocating function should not try to release the memory object if already released. The analysis requires to handle conditional consumers as well. These are functions that consume the memory object under certain conditions: consider a socket packet-send function that consumes the allocated socket buffer only in the case of a successful send. In such case, the caller of packet-send has to release the buffer if sending fails. K-MeLD employs an inter-procedural path-sensitive analysis to track memory object propagation and identify consumer functions.

We implemented our tool as multiple LLVM passes, rule mining, and rule application. We conservatively identified more than 800 memory allocation functions, most of which are specialized ones, and the associated release functions. After the ownership reasoning via inter-procedural analyses, and rule application, we detect 218 new memory leaks bugs even in many specialized modules. Our evaluation also confirmed that our ownership reasoning mechanisms significantly improves the accuracy.

4.3 Overview of K-MeLD

The goal of K-MeLD is to thoroughly and precisely detect memory leak bugs in kernel code. To identify a memory leak bug in a function, we model a memory leak as a case satisfying the following conditions:

1. A function retains ownership of the allocated memory
2. The function finishes without releasing the allocated memory

Originally, the allocating function is the owner of the memory object. However, the ownership of the object would likely propagate to other functions, e.g., the pointer to

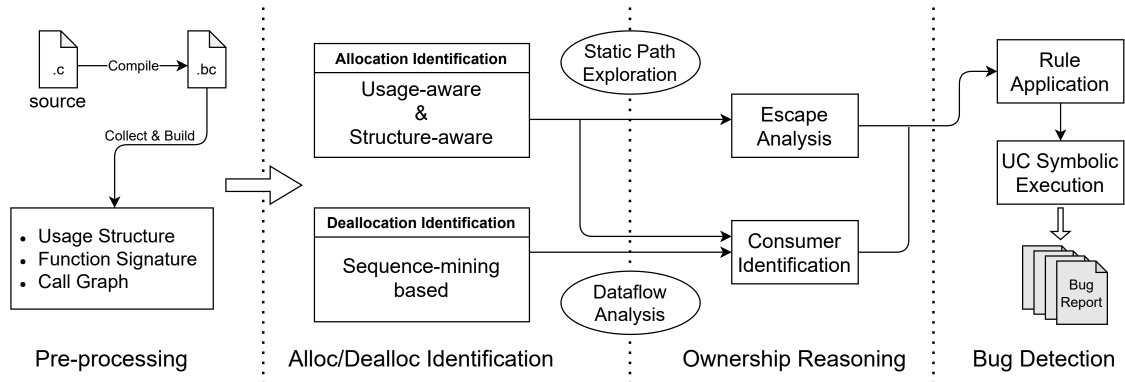


Figure 4.2: Overview of K-MELD. Identifying specialized allocation functions and the corresponding deallocatoin is followed by inter-procedural escape and consumer function analysis to determine the ownership of each allocation. Rule application checks the presence of deallocation and reports the bugs.

the memory object is passed to other functions (e.g. via return or parameters). As long as a function owns the memory region, the function is supposed to release the memory region; failure to do so is a memory leak. According to the modeling, we structure K-MELD into four phases, as shown in Figure 4.2: allocation/deallocation identification, ownership propagation analysis, and missing deallocation detection.

As shown in Figure 4.2, K-MELD compiles the source code into LLVM IR bitcode files. Such bitcode files are first preprocessed to extract contextual and structural properties. More specifically, we collect return type, argument signature, definition and usages of functions. At multiple points of our analyses, a call graph is used to identify the set of callees or callers for each function. Therefore in the preprocessing step, the global call graph is constructed via maintaining a map of function and all of its callees. To resolve indirect calls, we use function signatures to match the target function from among functions whose address is taken. The preprocessing phase is performed once.

Phase 1: allocation/deallocation identification. We identify potential allo-

cation functions by looking for pointer-returning functions that are followed by a null-check on the returned pointer. Additionally, we prune any function that is off-setting the returning pointer from an arguments. We also track the usage of the returned pointer to determine initialization before being de-referenced. Once the allocation functions are identified, deallocation functions are identified via rule mining. We refer to the current allocation function as the "function of interest" (FOI), while the function that calls the FOI is the "allocating function."

The high-level rationale behind using rule mining is that assuming the correct behavior is prevalent in a big code base like the OS kernel, we can model such common behavior in the form of sequential patterns. Rule mining uses sequential pattern discovery techniques to identify the sequence of commonly used operations on the memory object. This way for a specific FOI we can identify the common sequence of operations that are used for deallocation. For example, considering *kmalloc* as FOI, by looking through error handling paths of *kmalloc* call-sites we can observe the function *kfree* is called in most cases before terminating the execution path. Thus one can conclude that *kfree* is the associated deallocation function.

Phase 2: ownership propagation analysis. Recalling the two conditions proposed earlier in this section, condition (1) needs to know if the subject call-site is the owner of the allocated memory object or not. On one hand, the escape analysis tracks the propagation of the memory object upwards in call-graph beyond the allocating function boundaries. On the other hand, the consumer function detection tracks the propagation of memory object downwards in call-graph. These analyses determine if the subject call-site is still the owner of memory object. Such analyses are realized via a customized data-flow analysis. They need to be path-sensitive, to differentiate the conditional escape or consumption. They also need to be field-sensitive to be able to track the pointer propagation via struct fields. Such analysis tracks the def-use chain of an allocated memory object on each execution path. The escape

analysis determines if the memory pointer is copied beyond the allocating function (via a reference argument or global variable). The consumer function determines if any callees of the allocating function are releasing the memory object or causing it to escape. In either case, the allocating function is not the owner of the memory object anymore. Otherwise, the allocating function is responsible for appropriately releasing the memory object before returning.

Phase 3: missing deallocation detection. Once the previous phases finish, we can evaluate condition (2) for the specific FOI. That is any potential execution path missing appropriate release function is a potential memory leak bug. The context-aware Rule Mining identifies appropriate release functions associated with each FOI, and then escape analysis and consumer function detection analysis reason on the owner of allocated memory object. For each FOI, any absence of associated deallocation indicates a potential memory leak bug. These error-sites are then passed to under-constrained symbolic execution to confirm their feasibility.

4.4 Allocation and Deallocation Identification

4.4.1 Identifying Allocators

The first challenge K-MeLD overcomes is identifying allocation functions, both generic and specialized ones. Through the rest of this thesis we call such an allocator a Function Of Interest (FOI). Manual identification of allocation functions in an OS kernel is not practical. That is because various kernel modules have their own specialized allocation functions. We need an automated way to be able to collect a set of allocations including the specialized ones.

Observations. We observe that memory allocation is a critical operation, which in case of success returns a pointer to the allocated memory region, whilst in case

of failure it returns a NULL pointer. Because of this, the allocation functions are followed by a null-check on their return value. When a memory object is allocated, it requires initialization before being used effectively. An initialization is realized either via an `store` instruction or a call to `memcpy/memset` with the allocated pointer as destination. Such initialization must be performed before any read from the memory object.

To summarize, an allocation function has the following properties.

- It returns a pointer.
- The pointer is NULL checked before actually being used.
- The pointer is not derived from another base pointer.
- The object is initialized before being used, e.g., being read.

Removing getter functions. Besides the allocator functions, there is another class of functions that return a pointer and may be followed by a null-check. A *getter* function is a function that produces a pointer out of one of its pointer arguments either by indexing or accessing a field in struct. In order to exclude getter functions from the set of initial allocators, we first profile all pointer-returning functions in the kernel and mark those that their returning pointer is derived (calculated or accessed via `GetElementPtr`) from one of pointer arguments as base pointer.

Characterizing Pointers. We evaluate the aforementioned properties via use-finding and source-finding data flow. Use-finding is a forward data flow tracking that determines the operations on a pointer. For example use-finding helps to identify any null-check on a pointer. Source-finding is a backward data flow tracking that determines what is the source of pointer. This, for example helps in determining if a pointer is offset of a base pointer.

At this stage, our analysis favors recall over precision in terms of the number of null-check or initialization. More specifically, first we look for pointer returning functions which are not profiled as getter function (i.e. offset returning) and are followed by null-check and initialized in majority of their call-sites. This set will be further refined when deallocation detection is finished.

4.4.2 Context-aware Rule Mining for Deallocation Identification

After we identified allocation functions, we next identify the corresponding deallocation functions. To do so, we use sequential pattern mining on the error-handling operations.

Identifying and Collecting Error-Handling Paths.

It is a convention that failure of an operation in an OS kernel is reflected in a return status, which in turn should be checked by the caller. The error handling is responsible for recovering from errors and preventing the system from entering an unstable or undefined state. Error-handling paths are relatively short, and have clean and important operations to recover from the failure, this makes them a great place to identify paired operations (e.g. release, unlock or close).

To identify error handling paths at each FOI call-site, we first extract intra-procedural execution paths. To do so, we statically explore the control flow graph (CFG) in a depth-first fashion. Each path starts from the function's entry basic block and ends in a basic block with return instruction as terminator. Loops are unrolled for just one level. We also filter out the execution paths that do not go through the FOI call. In most cases the FOI return status is checked[5]. If the call to the FOI fails, the corresponding failure branch will not continue to use the resources (e.g., a

[5] If such a check is missing, it can be a missing-check bug. Since such bugs are out of the scope of this proposal, if FOI is not checked, we assume it was always successful.

pointer) returned by the FOI. Therefore, any paths following the FOI failure branch are not interesting to us because there will be no memory leaks. For example, the path going through line 15 of Figure 4.1 cannot lead to a memory leak of `skb`.

From the set of FOI success paths, we then identify error handling paths. If a path explicitly returns an error code (similar to those used in `errno` in user space, though kernel conventions use negative values) or NULL, it is an error-handling path. Not all error-handling paths are simple to identify. In most cases, the return status of an operation is checked, and if it turns out as an error code, the same error code is propagated up to the caller.

Referring to Figure 4.1, the return status of call to `crypto_report_alg()` is checked at line 24 and the same status code is returned. To identify such more complicated error-handling cases, we look for a critical check. We define a critical check as a check against zero or NULL that has no fall-through and leads to a return instruction with an integer or NULL parameter. A zero/non-NULL return status indicates that an operation was successful, while a non-zero (usually a negative errno) or NULL return status is an indication of the operation failure. Therefore, for each critical check, based on the check predicate we can determine which side of the check is taken when an error happens. This way we can identify non-explicit error handling paths like the one shown in Figure 4.1 line 24.

Sequential Pattern Mining. The mining is applied to the sequential patterns of operations on error-handling paths of a given FOI. This way we can extract the common patterns of error handling for each FOI. A key part of such a finding is which release functions are used to de-allocate the allocated memory object. In terms of memory management, it means when an allocation succeeds (i.e., the memory is allocated), if an error happens later, how the allocated memory should be released. This can be represented by a high-level rule in the form of ⟨*call FOI*, *check*, *release*, *return*⟩. Such a sequential pattern of operations is important because the OS kernel has differ-

ent mechanisms and functions to allocate and release memory dynamically. Finding associated release functions for any FOI can be addressed via looking for the sequential pattern of operations at the moment of error handling. Many specialized allocation functions allocate and craft specialized memory objects, which in turn require specialized de-allocation. Using the rule mining technique we identify such specialized release functions associated with a given FOI.

Frequent-pattern-mining algorithms expect the inputs in the form of a sequence of operations. Once we extract the error handling paths as described in Section 4.4.2, each path is transformed into a representation of sequential opcodes. To do so, we retrieve the LLVM opcode of each instruction on the path and form the opcode sequence. For the case of call instructions, we also retrieve the callee function name.

For each FOI, the opcode sequences are fed into the frequent pattern mining algorithm. The result will be a set of opcode patterns associated with error handling paths of the FOI. We take the most general pattern which by definition will be covering the maximum number of error handling paths and use it to identify the associated release function. Such a function is identified by cross-checking the result of mining against the set of operations as the last operation on the allocated pointer. Use-finding has a key role in here as we identify all operations that use a copy of the allocated pointer and consider only such operations in rule mining.

4.5 Ownership Reasoning

Another key challenge we have to address to detect kernel memory leaks is to decide where an allocated object is supposed to be released. In K-MeLD, we propose a new ownership reasoning mechanism to infer the release locations. The ownership reasoning mechanism includes two components: enhanced escape analysis and consumer function detection.

4.5.1 Enhanced Escape Analysis

Understanding how a function manages the ownership of allocated memory is a key factor in designing a precise memory leak detection technique. The second condition of resource release bugs is that the owner function of the allocated memory fails to release the allocated memory upon finishing the uses. If the allocating function passes on the ownership of the memory object, then there will be no leaking problem if the allocating function finishes without releasing the allocated memory. The ownership can be transferred either via returning the allocated pointer or assigning the allocated pointer to a global object or a reference argument. We call such ownership transfer escaping the pointer.

For example Figure 4.3 shows a case of an escaping allocated pointer. Here the allocated pointer escapes at line 33 via reference argument `bounce_buf_ret`. Looking at the call graph reveals that the function `hgcm_call_preprocess_linaddr()` is called repeatedly by `hgcm_call_preprocess()`. Eventually `bounce_bufs` is released by the caller of `hgcm_call_preprocess()` which is `vbg_hgcm_call()`. But the code in Figure 4.3 is still leaking memory at line 27, and it is because the pointer does not escape on this path. This shows the importance of path-sensitive escape analysis that we are employing to determine how the ownership of the allocation is changed. The leak can be resolved by moving the assignment at line 33 to line 23. When an allocated pointer is escaping on a path, then we assume there is another place of code responsible to manage the allocation, and the current path is not further explored for leak finding. This avoids the path-explosion problem and the tracking of complicated data flows, which in turn reduces false positives.

As mentioned in Section 4.4.1 source-finding is a backward data flow tracking which is employed to identify the sources of any destination that the allocation pointer is copied into. Source-finding determines that the destination of the copy at line 33 is a function argument. We identify escaping pointers as the pointers that their

```c
/* File: drivers/virt/vboxguest/vboxguest_utils.c */
static int hgcm_call_preprocess_linaddr(
	const struct vmmdev_hgcm_function_parameter *src_parm,
	void **bounce_buf_ret, size_t *extra)
{
	void *buf, *bounce_buf;
	bool copy_in;
	u32 len;
	int ret;

	buf = (void *)src_parm->u.pointer.u.linear_addr;
	len = src_parm->u.pointer.size;
	copy_in = src_parm->type != VMMDEV_HGCM_PARM_TYPE_LINADDR_OUT;

	if (len > VBG_MAX_HGCM_USER_PARM)
		return -E2BIG;

	/* Memory is allocated here */
	bounce_buf = kvmalloc(len, GFP_KERNEL);
	/* Check for allocation success */
	if (!bounce_buf)
		return -ENOMEM;

	if (copy_in) {
		ret = copy_from_user(bounce_buf, (void __user *)buf, len
	);
		if (ret)
			return -EFAULT;
	} else {
		memset(bounce_buf, 0, len);
	}

	/* Allocation pointer assigned to a pointer argument */
	*bounce_buf_ret = bounce_buf;
	hgcm_call_add_pagelist_size(bounce_buf, len, extra);
	return 0;
}
```

Figure 4.3: An example of escaping pointer: The allocation pointer `bounce_buf` is escaping via a pointer argument at line 33. But it is not enough to the prevent memory leak at line 27 (CVE-2019-19048).

ownership is passed to other functions, so it is safe if the current function does not release them. To identify if an allocated pointer is escaping we should find any variable that such pointer is copied into (via use-finding) and then determine the kind of the destination variable (via source-finding). If the destination variable is used in a return instruction, is a global variable, or is a function argument, then we can conclude that the allocated pointer escapes.

Path-sensitive escape analysis. Such an escape analysis should be path-sensitive as not all the execution paths may reach to the escaping point (as in Figure 4.3). Therefore, after identifying error handling paths (as described in Section 4.4.2) we perform escape analysis on each path. If the allocated pointer escapes on a specific error handling path, then we exclude such a path from the rest of our analysis, because the ownership is changed and the allocating function is not the sole entity having a handle on the allocated memory. It means if the allocating function terminates without releasing the allocated memory, there are still other live pointers to the allocation and no memory leak yet has happened. Instead we collect the escaping pointer information and go after the callers of the allocating functions one by one, and look for potential memory leak. This requires an inter-procedural analysis to track the operations performed on the escaping pointer in the context of the callers.

4.5.2 Consumer Function Detection

Consumer functions are another place that the ownership of an allocated memory object is changed. An allocating function may pass the memory object to its callees, so K-MeLD analyzes those receiving callees to make sure ownership is not changed once returning from such callees. If the callee is a consumer, then there will be no memory leak. It means any execution path that is going through a consumer function, should be disregarded for memory leak detection.

```
int cxgb4_get_srq_entry(struct net_device *dev,
                int srq_idx, struct srq_entry *entryp)
{
        ...
        struct adapter *adap;
        struct sk_buff *skb;
        ...
        adap = netdev2adap(dev);
        s = adap->srq;
        ...
        skb = alloc_skb(sizeof(*req), GFP_KERNEL);
        if (!skb)
                return -ENOMEM;
        ...
        t4_mgmt_tx(adap, skb);
        ...
        return rc;
}
```

Figure 4.4: Consumer function example: `t4_mgmt_tx()` consumes the buffer `skb` unconditionaly.

Figure Figure 4.4 shows an example of consumer function. The specialized allocator `alloc_skb()` allocates a new network buffer `skb` at line 12. At line 16 `skb` is passed to the function `t4_mgmt_tx()` which in turn passes `skb` to `ctrl_xmit()` at line 26. As the implementation of `ctrl_xmit()` shows the buffer is consumed on all execution paths. More specifically, the `skb` is released at lines 36 and 47, and is escaped via being added to a queue at line 42. This confirms the code at line 18 should not release `skb`. This example shows how K-MeLD requires an inter-procedural analysis to track `skb` across the function calls.

Remember at this stage K-MeLD knows associated deallocations of the FOI. So a pointer receiving callee becomes a consumer when on all of its execution paths it either releases or escapes the allocated pointer. This is realized by applying escape analysis to the paths of the callee, and also tracking the propagation of the allocated pointer in the callee to determine if it reaches any deallocation function.

Conditional consumers. While analyzing consumer function candidates, it may

```
/* File: net/dsa/tag_ksz.c */
static struct sk_buff *ksz_common_xmit(struct sk_buff *skb,
                struct net_device *dev, int len)
{
    ...
    nskb = alloc_skb(NET_IP_ALIGN + skb->len +
                padlen + len, GFP_ATOMIC);
    if (!nskb)
        return NULL;
    ...
    /* CONDITIONAL-CONSUMER */
    if (skb_put_padto(nskb, nskb->len + padlen))
        return NULL;
    ...
  return nskb;
}
/* File: net/core/skbuff.c */
int __skb_pad(struct sk_buff *skb, int pad, bool free_on_error)
{
  int err;
  int ntail;
  if (!skb_cloned(skb) && skb_tailroom(skb) >= pad) {
    memset(skb->data+skb->len, 0, pad);
    return 0;
  }
    ...
  if (likely(skb_cloned(skb) || ntail > 0)) {
    err = pskb_expand_head(skb, 0, ntail, GFP_ATOMIC);
    if (unlikely(err))
      goto free_skb;
  }
  err = skb_linearize(skb);
  if (unlikely(err))
    goto free_skb;
  memset(skb->data + skb->len, 0, pad);
  return 0;
free_skb:
    ...
    kfree_skb(skb);
  return err;
}
```

Figure 4.5: Conditional Consumer function example: `skb_put_padto()` consumes the buffer `skb` on failure.

be the case that the candidate consumes (releases or causes to escape) the memory object conditionally. For example in Figure 4.5 the allocated `nskb` is passed to `skb_put_padto()` which tries to pad the buffer by calling `__skb_pad()`. If it fails to do so, the memory object is released (as in line 39). But if it succeeds (as in lines 24 and 36) the ownership is not changed and the allocating function has to handle `nskb` appropriately. K-MeLD handles such cases by first identifying the critical check at line 12. Then via the condition predicate it determines if the current path is associated with success or failure of `skb_put_padto()` as described in Section 4.4.2. For example line 13 is associated with the failure, then when processing `skb_put_padto()`, K-MeLD only considers failure execution paths (only paths ending at line 40). This way K-MeLD determines the code at line 13 is not responsible to release `nskb`, and there is no leak.

4.6 Detecting Bugs Using Mined Rules

At this point, we have all the requirements to check the satisfiability of the two memory leak conditions in section Section 4.2: the function owns the allocated memory object, but fails to release it. As described in Section 4.4.1, we collect the initial set of FOIs and then extract the error-handling paths as explained in Section 4.4.2. These paths are fed into the rule mining to detect associated deallocation functions as described in Section 4.4.2. We prune the escaping paths and those going through consumer functions. The remaining paths are fed in a pattern-matching step to detect the paths that miss the release function. K-MeLD filters out paths that match the allocation-deallocation function pair. Furthermore, to prune any infeasible paths resulting from correlated branches, we apply under-constrained symbolic execution on each path that fail to satisfy patter-matching step.

```
1  /* File drivers/crypto/mediatek/mtk-aes.c */
2  static int mtk_aes_record_init(struct mtk_cryp *cryp)
3  {
4    struct mtk_aes_rec **aes = cryp->aes;
5    int i, err = -ENOMEM;
6
7    for (i = 0; i < MTK_REC_NUM; i++) {
8      aes[i] = kzalloc(sizeof(**aes), GFP_KERNEL);
9      if (!aes[i])
10       goto err_cleanup;
11
12     aes[i]->buf = (void *)__get_free_pages(GFP_KERNEL,
13             AES_BUF_ORDER);
14     if (!aes[i]->buf)
15       goto err_cleanup;
16         ...
17   }
18     ...
19 err_cleanup:
20   for (; i--; ) {
21     free_page((unsigned long)aes[i]->buf);
22     kfree(aes[i]);
23   }
24
25   return err;
26 }
```

Figure 4.6: An example of infeasible path introducing false positive.

4.6.1 Under-constrained Symbolic Execution

As introduced in Chapter 2, under-constrained symbolic execution is effective to analyze parts of the code. It serves to impose relaxed constraints on the inputs of a function by skipping the code preceding to the start of the function. This improves the scalability of the analysis.

Infeasible paths stemming from complicated data flow can be hard to avoid for static analysis tools. For example, Figure 4.6 presents an example where correlated control flows lead to a false alarm. The allocation at line 8 is happening inside a loop. If the code enters clean-up either via line 10 or 15, the loop at line 20 is responsible to release any previous allocations. Counter `i` determines if there exist such allocations

or not. A static control flow of this function may miss such correlation and assume a path reaching the `return` instruction at line 25 without going into the body of the `for` loop at line 20, leading to a false leak report.

To enable K-MELD to skip infeasible paths like the one presented in Figure 4.6, we adapted the under-constrained symbolic engine of Sys [10]. We customized the symbolic execution engine to be able to evaluate the feasibility of potentially leaking paths. Sys is only designed for intra-procedural symbolic execution, we enhanced the memory model to make it inter-procedural (detailed in Section 4.7.6). This way we were able to prune any infeasible path that were being reported as memory leak bug by K-MELD.

4.7 Implementation

We have implemented K-MELD as multiple LLVM passes, integrated with Python code to run the passes and rule mining, and then perform the pattern matching.

4.7.1 Potential Allocation Functions

As our ultimate goal is detecting memory leak bugs, we need to populate an initial set of potential memory allocation functions. Such functions will become the initial set of FOIs for the rest of our analysis. In order to scale to the large number of specialized allocation functions in the kernel, we need this phase to be automated.

Using a preprocessing LLVM pass over all the kernel, we identify any pointer-returning function. Then looking into the definition of that function, we make sure the returning pointer is not derived from any pointer argument, because an allocation function is supposed to return a previously non-existent memory object. To do so, we use source-finding described in Section 4.4.1 to make sure the returning pointer is a not coming from a pointer argument. After that, we look at the call-sites of the

candidate allocator and using use-finding analysis determine if the returned pointer is being null-checked and initialized or not. This can identify a caller function of a primitive allocator (like `kmalloc()`) as a new allocator. Such design choice reduces the tracking of complicated and lengthy data flows.

For some reasons not all memory allocations are null-checked. As an example if flag `GFP_NOFAIL` is passed to `kmalloc()`, the allocation cannot fail [56]. Additionally, some primitive allocators like `kzalloc()` or `kcalloc()` zero-initialize the memory object. Therefore, at this stage it is enough a candidate allocator to be null-checked or initialized at least in 40% of cases (such threshold was selected based on empirical observations in common allocators). This initial list of functions later will be pruned when we apply release detection described in Section 4.4.2. Any function that fails to yield at least one release function in rule mining will be discarded from the analysis results.

4.7.2 Path Exploration

To identify error handling paths, as described in Section 4.4.2, we statically explore the CFG and determine if a path is error handling or not. To avoid a path explosion problem, in addition to loop unrolling, we set a hard limit to the number of paths we explore per function. We also set a limit on the path depth to handle deep paths like mutually recursive calls. In the current implementation, we set the both limits to 1000. Such a limit was selected empirically to balance the pass running time and the leak detection rate.

When exploring the paths, we keep track of the uses of the allocated pointer. For each explored path, we maintain a list of opcodes that work on FOI result; specifically, function calls that take the FOI result as an argument. Such call instructions, are used for consumer function detection. For each explored path, we also record the last call instruction taking the FOI result as an argument in a set called *last_foi_use*.

4.7.3 Context-aware Rule Mining

For the purpose of sequential pattern mining, we employed the CloFAST [36] algorithm. CloFAST is a fast algorithm for discovering closed sequential patterns from a set of sequences. A sequential pattern is a sub-sequence that appears in many input sequences, and intuitively a pattern is closed if it cannot be extended. More precisely the *support* of a sequential pattern is the ratio of the number of times the sequential pattern appears divided by the total number of input sequences. The input to the CloFAST is a sequence database and a user-defined minimum support *minsup*. The sequence database is a set of sequences where each sequence is a list of opcodes associated with error handling paths as described in Section 4.4.2. The minimum support is a frequency threshold in terms of percentage which is used to recognize a frequent sequential pattern. A frequent sequential pattern is a pattern with a support level of no less than *minsup*. A closed sequential pattern is a frequent sequential pattern that is not a subset of any other pattern with the same support level. Discovering closed sequential patterns is more efficient than finding all sequential patterns while missing no information about the patterns [36]. Informally, as all sub-patterns of a frequent pattern are also frequent, therefore mining closed sequential patterns avoids generating unnecessary patterns and as a result yields savings of space and computational costs. In our implementation we empirically set the *minsup* to 0.6. In our implementation we noticed a limitation of the mining algorithm with respect to the number of input sequences. Meaning that the algorithm was not able to process all execution paths even on a machine with a large amount of memory. We avoided this limitation by first applying escape analysis to prune uninteresting paths, and then feeding in the sequence of operations on each path as input to the mining algorithm.

Release Function Identification. Once the rule mining is finished, we will have a set of sequential patterns for the specific FOI. We are interested in the patterns that

follow the correct behavior of memory release; meaning ⟨*Call FOI*, *check*, *release*, *return*⟩. Such a pattern is used to identify the associated release function to the FOI. We cross-check the mined pattern against *last_foi_use* set populated at the path exploration phase. This confirms the candidate release is the last function working on the FOI.

4.7.4 Ownership Reasoning

Path-sensitive Escape Analysis. We employ escape analysis to track the ownership of the allocated memory. K-MeLD's use of escape analysis is inter-procedural in that if a pointer escapes from the current function f, it is never reported as a leak from f. However all the callers of f are tracked to check for leaking the pointer.

Consumer Function Path Profiling. For the purpose of consumer function detection, K-MeLD analyzes any called function that takes the allocated pointer as an argument. It labels the callee paths as success or failure based on the return code. Then each caller path is associated with either of these path collections. If the caller path takes the success side of the consumer candidate, then callee's success paths are considered for releasing or escaping the allocated pointer, and vice versa.

Inter-procedural Data Flow. At multiple stages, K-MeLD employs inter-procedural data flow analysis to track the allocated pointer. For each LLVM call instruction taking allocated pointer as an argument, we determine the argument index in the call instruction. Then in the definition of the callee we start tracking the argument at that index. This way K-MeLD monitors the propagation of pointers across functions.

4.7.5 Pattern Matching

When we have identified the release functions associated with the specific FOI, K-MeLD then goes through the error handling paths extracted for the FOI and applies pattern matching. The patterns are of the form ⟨*call FOI*, *check*, *call release*, *return*⟩ where any call to the identified release functions will match the *call release* item.

Any path failing to match in the previous step will be further investigated for potential consumer functions. K-MeLD looks at the function calls on the allocated pointer and checks if the associated paths in the callee are consuming the pointer or not. If consuming, such a path is disregarded as a potential bug.

This way any error handling path deviating from the common approach of memory releasing will be found. Then such bug candidate paths are sent to under-constrained symbolic execution to identify and prune any infeasible path.

4.7.6 Under-constrained Symbolic Execution

Paths collected with pattern-matching step need to be reconstructed in the symbolic execution engine. To do so, we pass the function name and the basic block number associated with the errorsite. In the symbolic execution engine, we construct all paths from the function entry to the target basic block. Under-constrained symbolic execution engine examines these these specific paths for feasibility.

The symbolic execution engine in Sys [10] is intra-procedural. We needed inter-procedural under-constrained symbolic execution to confirm reachability of the buggy points along the call graph. To enable inter-procedural symbolic execution, we need the global call-graph (which is already constructed in pre-processing phase) along with the call site information. Using the call graph, we identify callers of the function with the potential leak. The call site information consists of the basic block pertaining the

call instruction and the list of function arguments. The calling basic block is required to construct paths from the caller function's entry point to the calling basic block. The list of arguments is required to let the symbolic engine construct inter-procedural path constraints. First the engine finishes symbolically executing the callee. Then for each feasible path we re-use the same symbolic state containing the path constraints and list of variables to construct path constraints in the caller. We only need to add an extra assignment on the function arguments to the local variables in the caller's context used in the call instruction. As an example, if function F has an argument x, when the caller makes a call to F and passes y as parameter to F, then the engine adds an assertion of the form $x == y$ when path constraints in the caller are formed.

Sys symbolic execution uses a linear memory model and assumes the pointers cannot alias. To enable inter-procedural symbolic execution we needed to make changes in the way the pointers are assigned. Once a pointer is allocated, Sys assigns a fixed location in the linear memory to that pointer. This causes problem when we want to carry constraints from the callee to the caller symbolic state. That is because the caller and callee pointers alias when an object is passed from the caller to callee. To solve this, we first maintain a map from the type of allocated pointers to the addresses assigned. When creating a new symbolic state for the caller, we use this map to *relax* locations that a pointer of a specific type may point to. This way, when we assert the equivalence of function arguments to the local variables in the caller, the SMT solver can resolve which address to use for the pointer.

These changes allowed us to use inter-procedural symbolic execution and prune infeasible paths to the potential leaking point in the code. Once the under-constrained symbolic execution determines that a path is feasible, we report it for manual audit. When we confirmed that it is a memory leak bug, then we prepare a patch to fix the bug and submit the patch to the code contributors.

4.8 Evaluation on the Linux Kernel

The effectiveness and scalability of K-MeLD is evaluated on the Linux kernel, version 5.2.13-stable. We compiled the kernel code into LLVM bitcode using `allyesconfig`, and got 18074 LLVM IR bitcode files.

4.8.1 Set of Allocations and Associated Deallocations

Using the methods described in Section 4.4.1 we populated an initial set of 4621 candidate allocator functions. Furthermore, once the rule mining finished, those failing to produce at least one associated deallocation, are pruned. This left us with 807 allocation functions[6]

These functions are considered as FOI for our evaluation. These FOIs have a wide range of frequency (some with many call sites, some with just a few), and so demonstrate K-MeLD's effectiveness for both general-purpose and specialized allocation functions. Withing this set of 807 functions, there are 4 generic allocation functions (like `kmalloc()`) with over one thousand callsites, along with more specialized allocation functions with callsites down to 2 (like `charlcd_alloc()`). K-MeLD was able to find the associated release functions for even the most specialized allocation functions (with as few as 2 callsites). Looking at the FOIs and associated releases, we identified only 15 false positive cases (1.8%). Moreover, there were allocators like `fscache_alloc_retrieval()` that were not selected as FOI, but were covered by K-MeLD. These functions are either escaping the allocation through pointer arguments, or did not pair with a specific deallocator. But essentially they are using other primitive allocators like kzalloc and are processed as escaping functions. Looking into the internal of 807 FOIs, we found a small number of primitive allocators (21 FOIs,

[6]The full list is available at: `https://github.com/Navidem/k-meld/blob/main/results/FOIs.txt`

comprising 2% of all selected FOIs) that are used to perform the actual allocations.

This selection of allocation and associated deallocation functions are necessary for effectively detecting memory leak bugs. To the best of our knowledge, none of the previous detection techniques used such a rich set of allocation-deallocation functions.

4.8.2 Found Bugs

In total, K-MeLD generated 373 leak warnings. After manual audit we confirmed 218 new memory leak bugs. This means K-MeLD is bearing 41% false positive. We reported the number of bugs based on the patches submitted, meaning that a single patch sometimes covers multiple memory leaks in the same function. 41 of memory leak bugs reported by K-MeLD received CVE IDs.

The bugs we found are spread among the most common allocation functions like `kmalloc()` (with 9244 call-sites) and `kzalloc()` (with 9926 call-sites); to the most specialized allocation functions like `sync_file_alloc()`, `edac_mc_alloc()` and `nlmsg_new()` with 2, 17 and 333 call-sites, respectively.

From the perspective of specialized modules, among 218 detected bugs, 115 were related to the FOIs with 400 call-sites or less. These results demonstrate the utility of K-MeLD to detect memory leak bugs even in specialized kernel modules.

4.8.3 False Positive Analysis

K-MeLD has a false-positive rate of 41%, which is acceptable for a static analysis tool applied to an OS kernel. We revisited the false warnings issued by K-MeLD to get an understanding on the sources of such false positives.

Customized device-managed allocations caused false warnings. These are driver-specific allocations that autonomously release all the allocated resources at the time of device detachment. Uncommon specialized release functions which K-MeLD fails to

identify in the rule mining step are another source of false positives. These functions do not pass the minimum threshold of mining and are basically not a wrapper for the common release functions; therefore failure to identify these functions contributes to the number of false alarms for K-MeLD. For example, K-MeLD correctly identified `kmem_cache_free()` as the release for `kmem_cache_alloc()`, but in 9 cases the allocated cache is released via `abort_creds()`.

4.8.4 False Negative Analysis

K-MeLD misses memory leak bugs if the allocation function is not in the set of FOIs. Additionally, even though we incorporate inter-procedural escape and consumer function analysis, these analyses are not complete and there may be cases where pointer propagation is lost due to aliasing or complicated data structure assignments. In order to perform a false negative analysis we decided to use 17 previously detected memory leak bugs which were assigned CVEs as ground truth. These bugs were spread among multiple versions of the Linux kernel over the span of 10 years. Instead of compiling multiple versions of the kernel, we reproduced each bug by applying the inverse of the proposed path to the same kernel version we were using for our experiments.

Table 4.1 shows the results of this analysis. Out of 17 cases, it turned out one was false positive and the initially merged patch was reverted later[7]. K-MeLD correctly identifies 9 of those memory leaks, and correctly identifies the reverted one as an escaped pointer. Two of those bugs are out of scope, as the source of the bug is not missing a release function, but API confusion[8]. Four of the bugs were not reproducible due to the code structure changes between kernel versions. Finally, K-MeLD missed one bug due to complicated pointer propagation. More specifically, the allocation

[7] CVE-2019-12379: commit 84ecc2f was reverted by 15b3cd8.

[8] The leak happens because `kvm_pin_pages()` expects size, but `kvm_unpin_pages()` expects the number of pages as argument.

CVE #	**Reproducible**	**K-MeLD Success/Failure**
CVE-2019-8980	✓	Success
CVE-2019-9857	✓	Success
CVE-2019-16995	✓	Failure
CVE-2019-16994	✓	Success
CVE-2019-15916	✓	Success
CVE-2019-15807	✓	Success
CVE-2019-12379	✓	Success (correctly rejected)
CVE-2018-8087	✓	Success
CVE-2018-7757	✓	Success
CVE-2018-6554	X	—
CVE-2016-9685	✓	Success
CVE-2016-5400	✓	Success
CVE-2015-1339	X	—
CVE-2015-1333	X	—
CVE-2014-8369	✓	out-of-scope
CVE-2014-3601	✓	out-of-scope
CVE-2010-4250	X	—

Table 4.1: False Negative Analysis results based on previous memory Leak CVEs

function causes the allocated pointer to escape by adding it to a list in a field of reference argument. K-MeLD loses the track of the pointer when it gets to the caller.

Chapter 5

Conclusion and Future Work

In this chapter we summarize what has been presented in this thesis and discuss potential future directions that can be taken for each of the projects.

5.1 Quantitative Information Flow Analysis

In quantitative information flow analysis the goal is measuring the amount of information flow from secret input to public output. We built upon an existing tool named FlowCheck [75] which employs a bit-level taint tracking along with accounting for implicit data flows. To measure the maximum information flow from the constructed data flow source to the sink, it looks for minimum-cut and returns the weight of such cut as the maximum amount of information leaked. FlowCheck is scalable but less precise compared to symbolic execution-based influence measurement, therefore we proposed a novel method to apply symbolic execution on smaller regions of the code to retrieve influence of the input data on the output of the region. Such tighter regional influence measurements will then update the global data flow graph which in turn affects the final maximum influence measurement. We devised multiple static and dynamic data tracking techniques along with an annealing-based region exploration to decide which code regions work the best for symbolically measuring the influence

over. Our tool named FCFB, showed its effectiveness on three use case binaries. In all of these cases FCFB provided a tighter yet correct total maximum influence from the secret input to the public output.

Possible future directions. Our FCFB prototype proved the effectiveness of the mixing data flow and symbolic execution idea. Furthermore, there are multiple potential improvements that can be evaluated. First of all, currently we are selecting just one region of code for symbolic execution. Expanding the regions excessively, increases the chance of symbolic execution failure due to increased path constraints complexity and as a result timeout. It is promising to consider how to choose multiple smaller regions for symbolic execution and update the global flow graph accordingly. Second, the current searching strategy is only benefiting from control flow structures for the purpose of region selection. This is while there are rich structures of data flow which can provide more help in terms of region selection. Third, the proposed symbolically influence measurement can become an optional region evaluation to vanilla FlowCheck. Currently, FlowCheck either analyzes a computation either without or with enclosure regions to contain the side effects of implicit flows. Symbolic influence measurement can be considered as a third method of flow evaluation for a computation like branch or loop structures. Last but not least, improving scalability of the underlying tool-chain to support more diverse set of instructions (like floating point instructions) can increase the applicability of such analysis tools.

5.2 Static Vulnerability Detection

Memory leak bugs are a serious security vulnerability in critical systems like OS kernels. In Chapter 4 we presented K-MeLD, an effective and scalable static memory leak detection tool for kernels that may consist of many modules. K-MeLD can detect memory leak bugs not only on general allocation functions, but also on spe-

cialized allocation functions with only a handful of call sites. K-MeLD first identifies allocation functions via a structure- and usage-aware approach, then associated deallocations are determined by using a sequential pattern mining technique. To detect memory leak bugs, K-MeLD reasons on the ownership of the allocated memory object to determine the location of expected dealocation call. Such reasoning is realized via inter-procedural and path-sensitive escape and consumer-function analysis.

Possible future directions. The general approach of rule mining employed in this paper can be applied to any resource release bugs. Even though we applied K-MeLD to memory allocation operations, the general idea of learning the correct sequence of critical operations from the majority of the code and looking for deviations is applicable to other resource acquisition operations. Another common case is locks: when a function acquires a lock, any potential error-handling path must release the lock. We leave such extensions for future work. As described in Section 4.7.6, we adapted under-constrained symbolic execution to eliminate infeasible paths to the buggy points. Such infrastructure can be used for exploitability verification. Generally, confirming a bug is exploitable or not is a very challenging problem. Dynamic approaches and fuzzing techniques try to start from the code entry points and detect bugs along their exploration of the big search space. Here, we can use the statically detected buggy point of code as a target and look for a path backward to the kernel entry points. Three major attacker-controlable entry points in the Linux kernel are system calls, interrupt handlers, and IO handler functions. A backward traversal on call-chain like the technique presented in [72] can be useful to identify feasible paths starting from the buggy point towards the kernel entry points. Specialized path selection mechanisms are required to prioritize which caller to take when moving backward on the call-graph. K-MeLD is also extendable to other OS kernels like FreeBSD. It requires compiling the source code into LLVM IR and then running K-MeLD on it to collect leak reports, and then auditing the reports.

References

[1] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976.

[2] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in Linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 255–268.

[3] J.-J. Bai, Y.-P. Wang, J. Yin, and S.-M. Hu, "Testing error handling code in device drivers using characteristic fault injection," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 635–647.

[4] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[5] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[6] J. C. Beatty, "Register assignment algorithm for generation of highly optimized object code," *IBM Journal of Research and Development*, vol. 18, no. 1, pp. 20–39, 1974.

[7] J. Bonwick *et al.*, "The slab allocator: An object-caching kernel memory allocator." in *USENIX Summer*, vol. 16. Boston, MA, USA, 1994.

[8] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting mobile communications: The insecurity of 802.11," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, 2001, pp. 180–189.

[9] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[10] F. Brown, D. Stefan, and D. Engler, "Sys: a static/symbolic tool for finding good bugs in good (browser) code," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 199–216.

[11] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[12] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the 17th ACM Conference on Computer and Communication Security (CCS)*, Chicago, IL, October 2010.

[13] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[15] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[16] B. M. Cantrill, "Method and apparatus for post-mortem kernel memory leak detection," Feb. 18 2003, US Patent 6,523,141.

[17] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy.* IEEE, 2012, pp. 380–394.

[18] 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. ACM, 2011, p. 5.

[19] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019, pp. 1707–1722.

[20] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 480–491.

[21] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, no. CONF, 2009.

[22] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012.

[23] J. K. Chittigala, "System and method for finding kernel memory leaks," Jul. 16 2013, US Patent 8,489,842.

[24] J. Cocke, *Programming languages and their compilers: Preliminary notes*. New York University, 1969.

[25] ——, "Global common subexpression elimination," in *Proceedings of a symposium on Compiler optimization*, 1970, pp. 20–24.

[26] G. Denaro, A. Margara, M. Pezze, and M. Vivanti, "Dynamic data flow testing of object oriented systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 947–958.

[27] G. Denaro, M. Pezze, and M. Vivanti, "On the right objectives of data flow testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 71–80.

[28] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 31–36.

[29] R. W. Eglese, "Simulated annealing: a tool for operational research," *European journal of operational research*, vol. 46, no. 3, pp. 271–281, 1990.

[30] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[31] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502041

[32] D. Engler and D. Dunbar, "Under-constrained execution: making automatic code destruction easy and scalable," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 1–4.

[33] C. Erickson, "Memory leak detection in embedded systems," *Linux Journal*, vol. 2002, no. 101, p. 9, 2002.

[34] M. Fleischer, "Simulated annealing: past, present, and future," in *Winter Simulation Conference Proceedings, 1995.* IEEE, 1995, pp. 155–161.

[35] A. C. Fong, "Generalized common subexpressions in very high level languages," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 48–57.

[36] F. Fumarola, P. F. Lanotte, M. Ceci, and D. Malerba, "CloFAST: closed sequential pattern mining using sparse and vertical id-lists," *Knowledge and Information Systems*, vol. 48, no. 2, pp. 429–463, 2016.

[37] A. Futoransky and E. Kargieman, "CORE-SDI-04: SSH insertion attack," Jun. 1998. [Online]. Available: https://marc.info/?l=bugtraq\&m=93656900402840

[38] "FuzzBall: Vine-based Binary Symbolic Execution," http://bitblaze.cs.berkeley.edu/fuzzball.html, accessed: 2021-05-10.

[39] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for C programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 459–470.

[40] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-Miner: Uncovering memory corruption in linux." in *NDSS*, 2018.

[41] P. Godefroid, "Higher-order test generation," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 258–269. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993529

[42] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[43] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[44] Google, "syzkaller - kernel fuzzer," 2019, https://github.com/google/syzkaller.

[45] C. Goues and W. Weimer, "Specification mining with few false positives," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, ser. TACAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 292–306. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00768-2_26

[46] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: Error handling is occasionally correct." in *FAST*, vol. 8, 2008, pp. 1–16.

[47] D. Henderson, S. H. Jacobson, and A. W. Johnson, "The theory and practice of simulated annealing," in *Handbook of metaheuristics*. Springer, 2003, pp. 287–319.

[48] W. E. Howden, "Symbolic testing and the dissect symbolic evaluation system," *IEEE Transactions on Software Engineering*, no. 4, pp. 266–278, 1977.

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

[50] A. W. Johnson, *Generalized hill-climbing algorithms for discrete optimization problems.* Virginia Polytechnic Institute and State University, 1996.

[51] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1913–1930. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/jung

[52] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *Proceedings of the 7th international symposium on Memory management.* ACM, 2008, pp. 131–140.

[53] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.

[54] K. Kennedy, *A survey of data flow analysis techniques.* IBM Thomas J. Watson Research Division, 1979.

[55] K. W. Kennedy, "Node listings applied to data flow analysis," in *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1975, pp. 10–21.

[56] Kernel.org, "kmalloc," 2019, https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html.

[57] ——, "Slab allocator," 2019, https://www.kernel.org/doc/gorman/html/understand/understand011.html.

[58] S. Kim and S. McCamant, "Bit-vector model counting using statistical estimation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2018, pp. 133–151.

[59] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[60] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[61] C. Koulamas, S. Antony, and R. Jaen, “A survey of simulated annealing applications to operations research problems,” *Omega*, vol. 22, no. 1, pp. 41–56, 1994.

[62] E. Larson and T. M. Austin, “High coverage detection of input-related security faults.” in *USENIX Security Symposium*, 2003.

[63] Y. Li, Z. Su, L. Wang, and X. Li, “Steering symbolic execution to less traveled paths,” *ACM SigPlan Notices*, vol. 48, no. 10, pp. 19–32, 2013.

[64] Z. Li and Y. Zhou, “Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code,” in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 306–315.

[65] B. Livshits and T. Zimmermann, “Locating matching method calls by mining revision history data,” in *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. ACM, 2005, pp. 296–305.

[66] E. S. Lowry and C. W. Medlock, “Object code optimization,” *Communications of the ACM*, vol. 12, no. 1, pp. 13–22, 1969.

[67] K. Lu, A. Pakki, and Q. Wu, “Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences,” in *28th USENIX Security Symposium*, 2019, pp. 1769–1786.

[68] K. Lu, C. Song, T. Kim, and W. Lee, “UniSan: Proactive kernel memory initialization to eliminate data leakages,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 920–932.

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

[70] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 103–116.

[71] lwn.net, "Injecting faults into the kernel," 2019, https://lwn.net/Articles/209257/.

[72] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *International Static Analysis Symposium*. Springer, 2011, pp. 95–111.

[73] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR.CHECKER: A soundy analysis for Linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.

[74] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 337–348, 2012.

[75] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 193–205.

[76] MITRE, "CWE-400: Uncontrolled resource consumption," 2019, http://cwe.mitre.org/data/definitions/400.html.

[77] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009, pp. 73–85.

[78] M. Orlovich and R. Rugina, "Memory leak analysis by contradiction," in *International Static Analysis Symposium*. Springer, 2006, pp. 405–424.

[79] OWASP, "Memory leak," 2019, https://www.owasp.org/index.php/Memory_leak.

[80] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys*, 2008.

[81] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity.* Courier Corporation, 1998.

[82] W. W. Peterson, W. Peterson, E. J. Weldon, and E. J. Weldon, "Error-correcting codes," *MIT press*, 1972.

[83] "ppm," http://netpbm.sourceforge.net/doc/ppm.html, accessed: 2021-05-10.

[84] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *29th International Conference on Software Engineering (ICSE'07).* IEEE, 2007, pp. 240–250.

[85] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[86] D. Rayside and L. Mendel, "Object ownership profiling: a technique for finding and fixing memory leaks," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* ACM, 2007, pp. 194–203.

[87] "rgb2gray," https://www.mathworks.com/help/matlab/ref/rgb2gray.html, accessed: 2021-05-10.

[88] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *ACM Sigplan Notices*, vol. 44. ACM, 2009, pp. 270–280.

[89] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[90] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 2013, pp. 1–12.

[91] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 74–83.

[92] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 15–30.

[93] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.

[94] V. Sharma, N. Emamdoost, S. Kim, and S. McCamant, "It doesn't have to be so hard: Efficient symbolic reasoning for CRCs," in *BAR 2020, Workshop on Binary Analysis Research*, 2020. [Online]. Available: https://dx.doi.org/10.14722/bar.2020.23011

[95] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.

[96] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 254–264.

[97] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1899–1913.

[98] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 163–177.

[99] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Proceedings of the 11th International Conference on Tools and*

Algorithms for the Construction and Analysis of Systems, ser. TACAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 461–476. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31980-1_30

[100] R. Williams, "A painless guide to crc error detection algorithms," *Internet publication, August*, 1993, Available: http://ross.net/crc/.

[101] 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*. Springer, 2015, pp. 135–154.

[102] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," in *Proceedings of ESEC-FSE*. ACM, 2005, pp. 115–125.

[103] 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*. ACM, 2015, pp. 414–425.

[104] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.