K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel

Zhengchuan Liang
UC Riverside
zlian064@ucr.edu

Xiaochen Zou
UC Riverside
xzou017@ucr.edu

Chengyu Song
UC Riverside
csong@cs.ucr.edu

Zhiyun Qian
UC Riverside
zhiyunq@cs.ucr.edu

***Abstract*—The severity of information leak (*infoleak* for short) in OS kernels cannot be underestimated, and various exploitation techniques have been proposed to achieve infoleak in OS kernels. Among them, memory-error-based infoleak is powerful and widely used in real-world exploits. However, existing approaches to finding memory-error-based infoleak lack the systematic reasoning about its search space, and do not fully explore the search space. Consequently, they fail to exploit a large number of memory errors in the kernel. According to a theoretical modeling of memory errors, the actual search space of such approach is huge, as multiple steps could be involved in the exploitation process, and virtually any memory error can be exploited to achieve infoleak. To bridge the gap between the theory and reality, we propose a framework K-LEAK to facilitate generating memory-error-based infoleak exploits in the Linux kernel. K-LEAK considers infoleak exploit generation as a data-flow search problem. By modeling unintended data flows introduced by memory errors, and how existing memory errors can create new memory errors, K-LEAK can systematically search for infoleak data-flow paths in a multi-step manner. We implement a prototype of K-LEAK and evaluate it with memory errors from syzbot and CVEs. The evaluation results demonstrate the effectiveness of K-LEAK in generating diverse infoleak exploits using various multi-step strategies.**

I. Introduction

As the trusted computing base for computer systems, the operating system (OS) kernel plays an important role in the security of computer systems. For this reason, OS kernels are major targets of attackers. Because popular OS kernels like the Linux kernel are mostly written in memory unsafe languages like C and assembly, exploiting memory errors (e.g., out-of-bound and use-after-free) remains the main attack vector against kernels. To mitigate such attacks, modern OS kernels have adopted several exploit mitigation techniques [6], [17], [19]. For example, the kernel address space layout randomization (KASLR) has made it difficult to launch reliable control-flow hijacking attacks. Consequently, lots of efforts have been dedicated to developing techniques that can circumvent these exploit mitigation mechanisms [31], [41]. One of such attacks is information leak (*infoleak* for short), which can be used to bypass KASLR [19] and disclose sensitive information such as cryptographic keys.

Information leaks are the consequences of exploiting software vulnerabilities to disclose randomized memory layout or sensitive contents of program memory [39]. There are two broad categories of approaches to exploiting OS kernels to leak information. One category is side-channel-based approaches, like micro-architectural side-channel attacks [22], [24]. The other category is memory-error-based approaches [33], where attackers exploit memory errors present in kernels to leak information. The second category is the focus of this paper. In general, memory errors allow the attackers to read and/or write the target program's memory in *unintended* ways [44]. For example, a spatial memory error like out-of-bound (OOB) allows attackers to access memory outside the boundary determined by the underlying memory object's type or allocation size. Similarly, a temporal memory error like use-after-free (UAF) allows attackers to access the memory object residing in the reallocated memory location. A successful kernel infoleak exploit leverages these unintended reads and writes as primitives to create a data-flow that propagates sensitive information to userspace [36], [38].

While infoleak is a critical step in bypassing KASLR and achieving other goals like privilege escalation, how to automate the generation of memory-error-based kernel infoleak has received little attention. The 2013 SoK paper [44] proposed a conceptual framework to reason about how memory errors can be exploited. Based on this conceptual framework, any memory error has the potential to be exploited to leak information. However, in practice, we only observe limited number of infoleak exploit strategies. Moreover, the conceptual framework is program agnostic, so it lacks modeling of program-specific behaviors and cannot be directly used to generated a concrete exploit. In a recent study [46], a more concrete reasoning framework is proposed for the Linux kernel. Unfortunately, since the main focus of it is to find kernel object that may be used in exploits and infoleak is not their focus, it also misses important information for generating infoleak exploits. Specifically, due to the large search space, generating infoleak exploits requires precisely modeling and reasoning of data-flow; but the data-flow analysis used in [46] is very imprecise.

Our work fills the gap by developing a novel graph-based data-flow reasoning and search framework. At a basic level,

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24935
www.ndss-symposium.org

we formulate the problem of crafting an infoleak exploit as searching for a series of data-flow fragments. Under this formulation, we offer several unique features that allow our solution to maximize the opportunity to identify infoleaks: (1) It unifies the handling of intended and unintended data-flow fragments that connect secret information to a leaking sink (e.g., `copy_to_user`). (2) It supports reasoning across the boundary of syscalls. (3) It allows the reasoning of the derivation of intermediate primitives (i.e., new memory errors) before finally discovering an infoleak. All of the above are cleanly packaged in our graph-based reasoning framework. Even though there can be multiple ways to achieve infoleaks, our solution can streamline the process through a unified graph search. For example, it easily handles the case where some sensitive information read by an unintended read primitive (UAF or OOB) propagates through multiple syscalls to a sink. It also handles the elastic-object-based infoleaks [14], where attackers corrupt the `length` field of an object[1] with an unintended write primitive first, and then cause an out-of-bound read of the data, which is then copied to userspace. In addition to finding infoleak paths, similar to Sleak [26], our framework can also track how the leaked info is transformed, and reconstruct its original value (e.g., a kernel pointer).

To validate our idea, we develop a prototype named K-LEAK, and apply it to test 250 real-world fuzzer-exposed memory bugs in the Linux kernel, i.e., syzbot bugs. K-LEAK is able to find infoleak paths in 21 bug reports (which are dynamically verified) through a variety of strategies, including simple single-syscall leaks as well as multi-syscall ones that start with either a read or write primitive. Many of our findings are not previously known. Based on the results, we developed seven end-to-end infoleak exploits to further demonstrate K-LEAK's effectiveness.

In summary, this paper makes following contributions:

- We propose a unified graph-based reasoning and search framework that enables us to identify infoleak paths in the Linux kernel. It has a number of unique features that maximize its opportunity to discover infoleaks.
- We develop a prototype solution called K-LEAK which leverages advanced static analysis to derive a specialized data-flow graph, amendable to infoleak reasoning. The source code can be found at https://github.com/seclab-ucr/K-LEAK for the purpose of reproduction of results and further research.
- We demonstrate the effectiveness of K-LEAK by evaluating it on a large-scale dataset of syzbot bugs. The results show that K-LEAK uncovers various exploit strategies, enabling us to find previously unknown infoleak opportunities.

II. Background

In this section, we introduce the necessary background to understand the rest of the paper. We start with a brief introduction to data-flow analysis and data-flow graph. Then we describe the points-to analysis and data-flow analysis we use in this work.

[1]These objects have variable sizes and hence called *elastic objects*.

A. Data-flow Analysis and Data-flow graph

Data-flow analysis is a body of techniques widely used in program optimization and analysis [9]. The primary goal of the data-flow analysis is to understand the flow of some property of interest within a program, e.g., the possible values of variables and expressions at different program points, as well as the relationships between them. Finding an infoleak exploit is a form of data-flow analysis—we aim to find (1) whether a piece of sensitive information (e.g., a kernel function pointer) can flow to user space, and if so (2) in what form (e.g., in its original value or has been transformed).

Data-flow analysis can be performed in many different ways. The most known way is using the monotone framework that combines a complete lattice and a space of monotone functions and applies the fixed-point algorithm [35]. Data-flow graph is another way to perform data-flow analysis. Such graph is used in analyzing the property of data dependency, which is an important research question [43], and benefits bug finding and other security research [29]. Data-flow graph (DFG, a.k.a., data dependence graph or value flow graph [43]) is a program representation that captures the data dependencies in programs. Such graph enables quick queries regarding data dependency, thus benefiting broad themes of program analysis and compiler optimization research [20], [29], [37], [43]. Regarding the goal of infoleak in this paper, DFG can help quickly answer the query of whether a piece of sensitive information can flow to user space. DFG can serve this purpose by modeling infoleak as sensitive information being passed on along the data-flow edges in DFG. Although the DFGs defined in previous works might have different appearances for different applications, they are fundamentally similar in terms of capturing program data dependencies. Usually, a node in DFG represents a program element (e.g. program statement and variable), and an edge represents an immediate data dependency. Since data dependency is viewed as data flow in the narrow sense, we will use data flow (analysis) instead of data dependency (analysis) in the rest of the paper.

However, DFGs of existing works assume the correct execution of program and only model intended data flows. They don't take into account the presence of memory errors, failing to model the resulting unintended data flows. The ability to model them is crucial in infoleak exploit generation in which memory errors are taken advantage of. To this end, we develop our version of DFG, named Memory-error-augmented data-flow graph (M-DFG) as the data-flow representation of the kernel in this work. It captures both intended and unintended data flows in the kernel. In M-DFG, nodes represent program elements, and edges represent both intended and unintended data flows among them. M-DFG are specifically designed for our iterative search algorithm, which searches for infoleak paths and controlled pointers that derive new memory errors. We describe M-DFG in detail in §III and §IV.

B. Cross-syscall Points-to Analysis and Data-flow Analysis

Points-to analysis is needed to track data flow through memory. It would be impossible to produce useful data-flow analysis results without points-to analysis [35]. Specifically, when we have two memory operations $l_1 : v_1 = *p_1$ and $l_2 : *p_2 = v_2$, an essential question in any data-flow analysis

is: whether v_1 and v_2 can be the same? To answer this question, we need to know whether p_1 and p_2 can be the same (i.e., aliasing). The analysis to answer it is points-to analysis, which attempts to determine what storage locations (i.e., points-to set) a pointer can point to [28]. Andersen's Algorithm [10] and Steensgaard's Algorithm [42] are two well-known algorithms to compute points-to results. After getting points-to analysis results, data-flow analysis can be done in the way we want.

Unfortunately, these points-to analysis and data-flow analysis algorithms cannot be directly applied to OS kernels as they assume the input is a complete program. However, the kernel has the characteristic of *multi-interaction* [13]. It has multiple program entry points since a user-level program can make multiple system calls (syscall for short) to interact with the kernel. As a result, data flows can propagate through multiple syscalls through global memory. In the context of infoleak, this means that a piece of sensitive information can be passed along the data flows that stretch across multiple syscalls. For instance, in syscall A, some sensitive information is stored to some global memory; later, in syscall B, the information is read from the global memory and propagated to some leaking sink. The multi-interaction nature of the kernel creates a large search space in complicating the reasoning of its data flows. For this reason, we need the ability to track data flow across syscalls.

SUTURE [51] is a static analysis framework for points-to analysis and data-flow analysis for the Linux kernel. It solves the multi-interaction problem using a summary-based approach. It performs points-to analysis and data-flow analysis for each kernel entry point (i.e., syscall entries) individually and produces points-to summary and data-flow summary for the entry. Both analyses are inter-procedural flow-, context-, field-sensitive. In its points-to analysis, it creates abstract memory objects for each entry and then uses access path to determine memory object aliases (e.g., memory object o_1 in entry A aliases with memory object o_2 in entry B) among entries. In its data-flow analysis, after obtaining the data-flow summaries for all entries, it "concatenates" the data flows of different entries into cross-syscall data flows.

SUTURE does not use Data-flow graph to perform the data-flow analysis. Instead it uses taint analysis (i.e., taint propagation) to reason about program data flows. Taint analysis works by propagating taints and check what program elements are tainted. In our work, for the points-to analysis part, we use the same approach as SUTURE. However, for the data-flow analysis part, instead of doing taint analysis, we make use of M-DFG, which is a more efficient data structure and more suitable for infoleak exploit generation, as will be discussed.

III. Overview

Problem Scope and Assumptions. In this work, we aim to automate the generation of memory-error-based infoleak against he Linux kernel. We consider exploiting side-channel vulnerabilities as out-of-scope. We assume attackers already know the existence of a kernel memory-error (i.e., a use-after-free or out-of bound vulnerability) and the corresponding bug triggering input (i.e., a bug reproducer program). This assumption is standard among existing automated exploit generation work [14] and can be satisfied by using a kernel address sanitizer (KASAN) report from syzbot as the starting point (Listing 1). The goal is to leak sensitive information out of the kernel (e.g., to user space or to network) by leveraging the initial memory error. During the generation of the exploit, we assume that attackers cannot hijack the kernel control-flow. We make this assumption because using infoleak to bypass KASLR is usually a prerequisite for kernel control-flow hijacking. Because automated kernel infoleak exploit generation itself is challenging enough, we assume the kernel *does not* employ any mitigation technique that regulates data-flow, such as data-flow integrity [11]. We leave such undeployed mitigation bypassing for future work.

```
BUG: KASAN: use-after-free in ax25_fillin_cb_from_dev
Read of size 4 at addr ffff8881ccecc438
Call Trace:
 ax25_fillin_cb_from_dev net/ax25/af_ax25.c:450 [inline]
 ax25_fillin_cb+0x6d5/0x810 net/ax25/af_ax25.c:477
 ax25_setsockopt+0x92a/0xa20 net/ax25/af_ax25.c:663
 __x64_sys_setsockopt+0xbe/0x150 net/socket.c:1910

The buggy address belongs to the object at ffff8881ccecc400
 which belongs to the cache kmalloc-192 of size 192
The buggy address is located 56 bytes inside of
 192-byte region [ffff8881ccecc400, ffff8881ccecc4c0)
```

Listing 1: KASAN report for the demonstrative example

Key Insight. The biggest hurdle to generate a memory-error-based kernel infoleak exploit is the huge search space—there are a massive number of data-flows in the kernel and most of them are not helpful towards an infoleak exploit. Existing approaches solve this problem by only looking for data-flow fragments or patterns that are known to be exploitable, such as "elastic objects" [14]. While this approach can effectively prune the search space, the drawback is also obvious: not all memory errors can be exploited in this way to achieve infoleak, as will be seen from our evaluation results §VII. Therefore, we aim to design a more general approach. Conceptually, finding an infoleak path is a typical data-flow analysis problem (e.g., similar to detecting potential privacy leaks in mobile apps [34]). However, existing data-flow analyses cannot be directly applied in searching for memory-error-based infoleak exploit because they lack the modeling of unintended data-flows introduced by memory errors. This leads to the key insight behind our approach—*by additionally modeling the unintended data-flows introduced by memory errors, we can utilize existing data-flow analyses, especially efficient ones, to generate infoleak exploits.*

Technical Challenges. While our idea may sound straightforward, realizing it faces a few technical challenges.

(1) *Modeling unintended data-flow*. The first challenge we need to address is how to model unintended data-flow introduced by memory errors. Our observation is that, memory errors are essentially dereferences of invalid pointers (i.e., pointers that are out-of-bound or pointing to freed/uninitialized memory). From the data-flow analysis perspective, the effect of an invalid pointer is enabling new data-flow between memory load (read) and store (write) operations. In other words, an invalid pointer enables unintended *aliasing* between pointers used in load and store operations. Based on this observation, we design a special alias analysis for invalid pointers §IV-D.

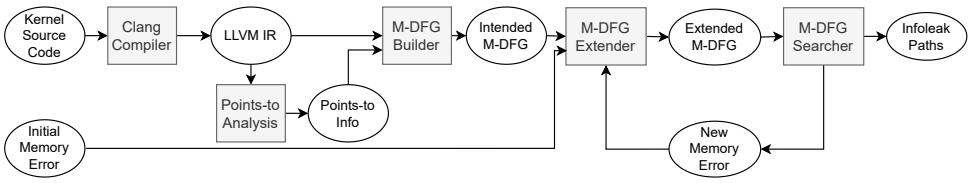

Fig. 1: High-level workflow of K-LEAK

(2) *Modeling data-flow across system calls*. One particular challenge in generating kernel infoleak exploits is that the exploit may involve multiple syscalls. The data-flow can cross the boundary of syscalls, in an intended way or unintended way. Intended cross-syscall data-flows result from the multi-interaction characteristic of the kernel, as has been described in §II-B. Without the ability to model them, some memory errors may not be exploitable [51]. We address this challenge by leveraging the cross-syscall point-to analysis from [51], as well as our DFG-based cross-syscall data-flow analysis. Unintended cross-syscall data-flows occur when a syscall unintendedly writes to or reads from memory that is used by other syscalls through memory errors. Such cases are handled by our solution to technical challenge (1).

(3) *Modeling additional memory errors*. In many cases, a single memory error may not directly be exploitable to achieve infoleak. For example, in elastic-object-based exploits, attackers first exploited an unintended write operation to overwrite the `length` field of an elastic object. This introduces a new out-of-bound read memory error when the elastic object is accessed. Specifically, the corrupted `length` field will influence how far a data pointer can reach during a `copy_to_user()` call. We address this challenge by designing an iterative search algorithm, where in each iteration, besides looking for potential infoleak path(s), it also searches for pointers that may be controlled by attackers. We then check whether those attacker-controlled pointers can derive new memory errors. If so, unintended data-flows introduced by these newly derived memory errors will be used in the next iteration for exploit generation.

Workflow. Figure 1 depicts the high-level workflow of K-LEAK. In the figure, a rectangle represents an analysis component that takes in input data and produces output data, and an oval represents data (input data or output data for analyses). K-LEAK takes as input the source code of the Linux kernel and an initial memory error (i.e., a KASAN memory error report along with the corresponding triggering input), and outputs potential infoleak data-flow paths. The central piece of our approach is a graph-based representation of data-flow named Memory-error-augmented data-flow graph (M-DFG). M-DFG is constructed by first extracting (done by `M-DFG Builder` in the figure) the intended data-flows from the kernel LLVM IR, then extended with unintended data-flows introduced by memory errors (done by `M-DFG Extender` in the figure). Potential infoleak data-flows are generated using an iterative search. In each iteration, K-LEAK leverages an efficient graph search algorithm (done by `M-DFG Searcher` in the figure) to find (1) potential infoleak paths and (2) pointers controllable by attackers. If new memory errors are found, M-DFG is further extended with new unintended data-flows before the next iteration. The iterative search stops when a threshold is reached (e.g., after some iterations or enough infoleak paths are found).

```
1  /********** [entry 1] ax25_setsockopt **********/
2  int ax25_setsockopt(struct socket *sock, int level,
3    int optname, char __user *optval, unsigned int optlen) {
4    struct sock *sk = sock->sk;
5    ax25_cb *ax25 = sk_to_ax25(sk);
6    ax25_fillin_cb(ax25, ax25->ax25_dev);
7  }
8  void ax25_fillin_cb(ax25_cb *ax25, ax25_dev *ax25_dev) {
9    ax25_fillin_cb_from_dev(ax25, ax25_dev);
10 }
11 void ax25_fillin_cb_from_dev(ax25_cb *ax25, ax25_dev
   ↪  *ax25_dev) {
12   ax25->n2 = ax25_dev->values[N2]; // UAF read. Both ax25
   ↪  and ax25_dev are heap pointers
13 }
14
15 /********** [entry 2] mon_bus_init **********/
16 void mon_bus_init(struct usb_bus *ubus) {
17   mbus->u_bus = ubus; // mbus is a heap pointer
18 }
19
20 /********** [entry 3] ax25_getsockopt **********/
21 int ax25_getsockopt(struct socket *sock, int level,
22   int optname,  char __user *optval, int __user *optlen) {
23   int val;
24   void *valptr = (void *) &val; // valptr is a stack pointer
25   length = min_t(unsigned int, maxlen, sizeof(int));
26   struct sock *sk = sock->sk;
27   ax25_cb *ax25 = sk_to_ax25(sk);
28   val = ax25->n2; // ax25 is a heap pointer
29   return copy_to_user(optval, valptr, length) ? -EFAULT : 0;
30 }
```

Listing 2: Kernel code for the demonstrative example

A Demonstrative Example. Here we use an example to demonstrate the basic idea of our K-LEAK approach. In this example, the starting point is a UAF memory error [7] found by syzbot. Listing 1 shows the KASAN report from syzbot, and Listing 2 shows the corresponding buggy code for which the report is produced. In function `ax25_fillin_cb_from_dev()` in Listing 2, the heap object that `ax25_dev *ax25_dev` points to is already freed. When `ax25_dev` is dereferenced at line 12, there is a UAF read. Then the read value is stored to a `ax25_cb` object that `ax25_cb *ax25` points to.

In order to exploit such memory error to achieve infoleak, an attacker can reallocate a heap object `struct mon_bus` (see `mon_bus_init()` in Listing 2) to reoccupy the location of the free `ax25_cb` object. The reallocated heap object contains sensitive information (field `u_bus`, a pointer) at the same offset as `offsetof(ax25_dev, values[N2])`. After the reallocation,

when the UAF read is triggered at line 12, the value of sensitive pointer (u_bus) will be read out and stored to ax25_cb object. In addition, the attacker needs to invoke getsockopt() system call that reaches entry ax25_getsockopt(). It first reads out the sensitive pointer from ax25_cb object at line 28, and then it copies it to user space at line 29.

Figure 2 shows the corresponding M-DFG for the demonstrative example. The horizontal path in the figure is the infoleak data-flow path that leaks the sensitive pointer to user space. The infoleak path consists of three segments (Figure 2 has three big rectangle areas), each belonging to a separate syscall entry. In the first entry when the kernel is executing mon_bus_init(), there is a store node that stores the sensitive pointer to object struct mon_bus (corresponding to line 17). There are two incoming edges for a store or a load node: data edge and pointer edge. A data edge (solid line in Figure 2) represents the data to be stored / loaded; the pointer edge (dash line in Figure 2) represents the address to store / load. Here, bus is stored to address &mbus->u_bus (some location inside object struct mon_bus). In ax25_setsockopt(), the load node (corresponding to line 12 where UAF happens) loads some memory content from address &ax25_dev->values[N2]. Since &mbus->u_bus and &ax25_dev->values[N2] alias due to reallocation, the load nodes actually loads the content that the previous store node stores. Therefore there is a data edge (the edge in red) linking the store node to the load node. This is also called read-after-write dependency. The edge is marked red because it is an unintended edge caused by a memory error (i.e., UAF). Similar idea applies to the data edge between the store node and copy_to_user node (a special kind of load node) in the third segment in Figure 2. Corresponding to line 28, a temporary register value is stored to a stack address valptr=&val; and the copy_to_user copies the content at address valptr to user space. Therefore, there is a data edge linking the store node to the copy_to_user node. The edge is not marked red because it an intended edge (i.e., not caused by any memory error).

Note that in traditional data-flow analyses, the red read-after-write data edge will not exist because they do not model memory errors. K-LEAK is able to capture this data-flow due to our memory-error-aware aliasing analysis §IV-D, which understands that object ax25_cb could alias with struct mon_bus because of UAF.

In summary, the attacker's goal is to follow all the solid edges (i.e., data edges) in Figure 2 to find an infoleak path, starting from sensitive information and ending at leaking sink.

IV. Memory-error-augmented data-flow graph

M-DFG is the central data structure in our framework to generate infoleak exploit. In this section, we define M-DFG and describe how it is constructed.

A. Pre-Analysis

As shown in Figure 1, M-DFG is constructed based on two inputs: (1) the Linux kernel in LLVM's partial static single assignment (SSA) form, and (2) the point-to information generated by a flow-, context-, and field-sensitive inter-procedural static analysis [51]. We consider most LLVM instructions when constructing M-DFG, including:

- Binary operations (BinOp): $v_3 = v_1 \oplus v_2$.
- Conversion operations (ConOp): $v_2 = (\tau)v_1$.
- Phi operation (Phi): $v = \phi(\ell_1 : v_1, \ldots, \ell_n : v_n)$.
- Memory access: (Load) $v = *p$, and (Store) $*p = v$.
- Addressing operations (AddrOf): $p = \&v$ (alloca), $p = \&v.f_i$ (struct field), and $p = \&v[idx]$ (array element).
- Function calls and returns (Call): $ret = f(v_1, \ldots, v_n)$.
- Instructions unrelated to data-flows like icmp and br are not supported.

Following prior conventions, we use abstract memory objects to represent distinct memory regions. Let $\mathcal{V} = \mathcal{A} \cup \mathcal{T}$ be all variables in LLVM, $\mathcal{A}$ be possible point-to targets, and $\mathcal{T}$ be all top-level variables whose address is not taken. To provide field-sensitive, addresses of different fields in a C/C++ struct is considered as distinct memory regions. For arrays, all elements are considered as a single memory region. The points-to information includes the set of abstract memory objects, and the set of points-to relations.

- An *abstract memory object* $o \in \mathcal{A}$ is an abstract storage location in static point-to analysis. They are field-sensitive address-taken variables (i.e., $\&v, \&v.f_i, \&v[0]$). For each object, we track how it is allocated (i.e., on stack or in heap) and its allocation site.
- A *points-to relation* relates a top-level variable to a set of abstract memory objects it may point to $v \mapsto o \in \mathcal{T} \times \mathcal{A}$; or from a abstract memory object to another abstract memory object $po \rightsquigarrow o \in \mathcal{A} \times \mathcal{A}$.

To preserve the context-sensitivity of the point-to analysis results, when constructing M-DFG, we clone each LLVM instruction ℓ and associate it with a context string ℓ^{ctx}. For abbreviation, we use $\boldsymbol{\ell}$ to denote instruction with context string. We use the memory usage of the static analysis as a threshold to determine the maximum context level. In our evaluation, we used 50GB as the threshold, and the corresponding context level is 7.

B. Graph Definition

Memory-error-augmented data-flow graph (or M-DFG for short) $G = (N, E)$ is a directed graph. Its nodes N represent program elements, and its edges E represent data dependencies between program elements. There are three kinds of nodes in M-DFG, representing different kinds of program elements in the partial SSA form:

- A *variable node* represents a definition of a top-level variable v in the program.
- A *load node* represents a memory read (Load) instruction in the program.
- A *store node* represents a memory write (Store) instruction in the program.

Note that we choose to explicitly model Load and Store instructions to assist identifying new memory errors (§V-C) and updating M-DFG with memory-error-induced data dependencies.

Edges E are directed. An edge $\boldsymbol{\ell}_1 \rightarrow \boldsymbol{\ell}_2$ represent data dependencies: $\boldsymbol{\ell}_1$ defines one or more variables used in $\boldsymbol{\ell}_2$.

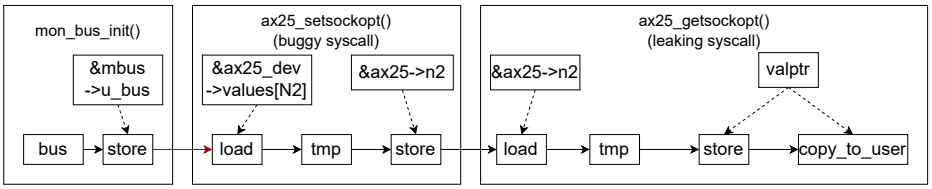

Fig. 2: Part of the M-DFG for the demonstrative example

To facilitate the identification of new memory errors, we distinguish two kinds of edges in M-DFG, data edges and pointer edges.

- A *data edge* represents a direct data-flow (def-use) that transfers data from the source node to the destination node (e.g., solid edge in Figure 2).
- A *pointer edge* represents a special data-flow only applicable to a LOAD or a STORE instruction, that is, the memory location depends on the source code (e.g., dashed edge in Figure 2).

C. M-DFG Builder

With the intended points-to information pre-computed, the intended M-DFG is built by creating nodes and link them with edges. This is done by the `M-DFG Builder` in Figure 1. Nodes and edges are added according to the rules in Table I. Recall that LLVM instructions are cloned and associated with a context string when added to the graph. Following the rules, we build a graph for each syscall entry separately, which will be merged into the whole graph later. Following are the detailed descriptions of the rules.

For BINOP, CONOP, and PHI instructions, nodes are created for each defined high-level variable v. Then data edges are added according to the standard def-use relation.

Our analysis is inter-procedural. For each function definition $f(\dots, p, \dots) \rightarrow r$, nodes are created for each formal argument p and the return value r. For each call site $\boldsymbol{\ell} : ret = f(\dots, a, \dots)$, nodes are created for each actual argument a and the variable ret to receive the return value. Then data edges are added from each actual argument to the corresponding formal argument $a \rightarrow p$, and from the return value to the variable that receives the return value $r \rightarrow ret$. To reduce false positives, we use dynamic tracing to resolve indirect calls (see §VI for details).

For a LOAD instruction $\boldsymbol{\ell} : v = *p$, nodes are created for the defined value variable v, and the load instruction $\boldsymbol{\ell}$ itself. A data edge is added from the load node to the defined variable $\boldsymbol{\ell} \rightarrow v$ to capture the data flow, and a pointer edge is added from the pointer variable node to the load node $p \rightarrow \boldsymbol{\ell}$ to facilitate discovery of new memory errors. To detect infoleak, `copy_to_user(to,from,len)` and a number of other APIs that perform load/store-like memory operations are treated as special kind of load/store nodes. Nodes are created for $from$, len and `copy_to_user`. Then two pointer edges are added from $from$ and len to copy_to_user.

For a STORE instruction $\boldsymbol{\ell} : *q = v$, a node is created for the store instruction $\boldsymbol{\ell}$. Then a data edge is added from the value variable node to the store node $v \rightarrow \boldsymbol{\ell}$, and a pointer edge from the pointer variable node to the store node $q \Rightarrow \boldsymbol{\ell}$.

To model data-flow through abstract memory objects, we add data edges from store node to load node according to the read-after-write (RAW) rule. Specifically, if the pointer variables of the store $\boldsymbol{\ell}_s : *q = v$ and pointer variable of the load $\boldsymbol{\ell}_l : v = *p$ are aliases (i.e., may *point to* the same abstract memory object o), then a data edge is added from the store node to the load node $\boldsymbol{\ell}_s \rightarrow \boldsymbol{\ell}_l$.

A special note is that only things present in LLVM are handled and things on stack (e.g., stack pointer, frame pointer) are not; Moreover, to reduce potential false positives introduced by static analysis, when building M-DFG, we only analyze syscall entries that can are covered by existing test cases from syzbot and those in published works [13]–[15] (more details in §VI).

Model cross-syscall data-flow. To address technical challenge (2), we model intended cross-sycall data-flow in M-DFG as follows. We connect the M-DFG of each syscall using cross-syscall RAW edges. Using the cross-syscall points-to information generated by SUTURE [51], we look at all pairs of syscall entries and try to apply RAW rule to connect the store node of one syscall entry to the load node of another.

D. M-DFG Extender

One main innovation of K-LEAK is modeling both intended and unintended data-flow in M-DFG. Specifically, RAW edges are used to model the data-flow through both intended and unintended memory accesses, which result from the execution without and with memory error, respectively. The key challenge, as mentioned before in §III, is that given a memory error, what RAW edges should be added.

An unintended memory access occurs when the pointer variable node becomes invalid and points to some unintended memory location (e.g., out-of-bound or freed). The question, therefore, is what unintended memory location the invalid pointer can point to. Traditionally, this question is answered by a point-to analysis like SUTURE [51]. However, existing point-to analysis only models correct execution semantics, thus cannot be used to answer this question. Therefore, we developed a *memory-error-aware* point-to analysis to answer this question.

From a high level, an out-of-bound (OOB) pointer allows attackers to access data in "adjacent" objects. Therefore, to answer the question of what abstract memory object an OOB pointer can point to, we need to know (1) what memory object may be allocated adjacently and (2) how far away the pointer can go OOB, or in other words, what is the *capability* of the

TABLE I: Rules for adding nodes and edges into M-DFG.

Instruction	Program Statement (SSA)	Nodes	Data Edges	Pointer Edges
BINOP	$\boldsymbol{\ell} : v_3 = v_1 \oplus v_2$	v_1, v_2, v_3	$v_1 \rightarrow v_3, v_2 \rightarrow v_3$	-
CONOP	$\boldsymbol{\ell} : v_2 = (\tau)v_1$	v_1, v_2	$v_1 \rightarrow v_2$	-
PHI	$\boldsymbol{\ell} : v_d = \phi(\boldsymbol{\ell}_1 : v_1, \ldots, \boldsymbol{\ell}_n : v_n)$	$v_d, v_1, \ldots, v_n$	$v_i \rightarrow v_d, i \in \{1, \ldots, n\}$	-
LOAD	$\boldsymbol{\ell} : v = *p$	$v, p, \boldsymbol{\ell}$	$\boldsymbol{\ell} \rightarrow v$	$p \Rightarrow \boldsymbol{\ell}$
STORE	$\boldsymbol{\ell} : *q = v$	$v, q, \boldsymbol{\ell}$	$v \rightarrow \boldsymbol{\ell}$	$q \Rightarrow \boldsymbol{\ell}$
CALL	$\boldsymbol{\ell}_c : ret = call f(\ldots, a, \ldots)$ $f(\ldots, p, \ldots) \rightarrow r$	a, p, r, ret	$a \rightarrow p, r \rightarrow ret$	-

TABLE II: Rule to add read-after-write data edges in M-DFG.

Rule	Program Statement	Nodes	Data Edges	Pointer Edges
RAW	$\boldsymbol{\ell}_s : *q = v \ [q \mapsto o]$ $\boldsymbol{\ell}_l : v' = *p \ [p \mapsto o]$	-	$\boldsymbol{\ell}_s \rightarrow \boldsymbol{\ell}_l$	-

OOB pointer. Similarly, for a dangling pointer, what abstract memory objects it can point to depends on what kind of object may be reallocated into the same memory region. Similar to prior work, we need to understand the behaviors of the heap allocator in order to pair the capability with appropriate objects [13], [46].

To answer the first question about memory objects, we rely on previously summarized objects in published work [13]–[15]. Specifically, we make the assumption that the syscalls capable of allocating and utilizing these objects are provided. By adopting this approach, we can statically conduct a memory-error-aware points-to analysis and generate aliases by incorporating new *pointer edges* into the graph (later in §V we also talk about how to dynamically verify the aliases).

To answer the second question about capabilities, we leverage the KASAN report to extract the required information. In other words, we prefer concrete capabilities over potential capabilities. For example, Listing 1 is a sample bug report, which shows that a memory read reads the 56 bytes inside the use-after-free slab cache slot. Listing 2 shows the line where the use-after-free happens. From both we know the pointer variable `&ax25_dev->values[N2]` points to the use-after-free memory location. We can assume the attacker can reallocate some object at the UAF slot to create unintended aliases. We update the points-to set of the pointer variable, letting it point to all objects that can be reallocated in the UAF location. By doing so, we can apply RAW rule to add unintended RAW edges. In the example, when the attacker reallocates an object `struct mon_bus` in the use-after-free slot, the store node that stores the pointer will be linked to the load node at line 3. As we can see, when the memory error (e.g. UAF and OOB) occurs at kernel heap, we assume the success of using heap manipulation technique [49], and update the points-to set for the pointer variable to be all objects allocated in the same slab spot (Elastic objects are considered in multiple slab caches). We check each object in the set of abstract memory objects in points-to information (described in §IV-A).

In cases where we can confidently confirm that the capability of a pointer is *arbitrary* (i.e. attacker can fully control the value of the pointer), we follow prior conventions from [14], [46]: for arbitrary read, we assume it can pair with all locations inside all objects and add corresponding RAW edges; for arbitrary write, we assume it cannot pair with any objects. This is because when the `panic_on_oops` kernel configuration is off (by default in popular distributions like Ubuntu), the kernel will only kill the current process even if an illegal memory access has caused a page fault in the kernel space. Therefore, an attacker can exploit an arbitrary read to continuously probe the address space. However, arbitrary write may corrupt kernel data structures and lead to unwanted side-effects.

V. SEARCH ON M-DFG

In this section, we describe the benefits of M-DFG, how M-DFG can be used to search for infoleak and new memory errors, and our iterative algorithm on M-DFG that combines the two searches and systematically covers the search space for infoleak exploits. With M-DFG, we are able to reduce infoleak exploit generation problem to a graph search problem. The graph searcher in Figure 1 performs the searches and applies the iterative algorithm. Intended points-to information, intended M-DFG and an initial memory error are prepared as initial inputs, and M-DFG will be updated per iteration.

As a reminder, we believe precision (i.e., reducing false positives) is more important than recall (i.e., reducing false negatives) for exploit generation; so, when there is a trade-off, we prefer precision over recall. For example, when building M-DFG, we only analyze kernel code that can be covered by existing test cases (i.e., syzbot proof-of-concept (PoC)) or those already analyzed by [13]–[15]. And when extending M-DFG with new memory errors, we only consider kernel objects that can be allocated by [13]–[15].

A. Benefits of M-DFG

Comparing to existing data-flow analysis solutions, our M-DFG solution has benefits in infoleak generation.

SUTURE [51] solves the data dependency problem with taint analysis, instead of building DFG data structure. It maintains and propagates taint records, and creates taint summary (i.e. from what taint source to what taint sink). However, taint

summary is not an efficient data structure in expressing data flows. It is basically a path-like data structure. The number of taint summaries and taint records can grow exponentially and incur large memory footprint; Also, it is not efficient to do data flow query on top of such summary. In contrast, M-DFG allows us to use multiple graph algorithms to reason about the data flows.

Other DFGs from existing work like [43] usually has variable nodes to represent top-level variables and memory nodes to represent memory locations. M-DFG removes memory nodes and adds load nodes and store nodes, as well as pointer edge that links a pointer variable node to a load or store node. We argue that both representation of data-flows are equivalent in expressing data flows. Specifically, we use `stored_value->store_node->load_node->loaded_value` to model read-after-write; others use `stored_value->memory_node->loaded_value` to model the same read-after-write. However, the biggest advantage of our representation is being able to reason about creating new memory errors. This is because our M-DFG design has pointer edge while existing DFG does not. This helps us to reason about additional memory errors by tracking whether the attacker can control a load or store node through a pointer edge, which will be discussed in §V-C. Moreover, our representation makes it easier to model the unintended data-flow resulting from memory errors and to perform the two searches.

Finally, because searching for infoleak and searching for new memory errors can both be modeled as data-flow analysis over M-DFG, in each iteration, we only need to search the graph once, instead of searching twice (one for infoleak and one for new memory errors).

B. Infoleak Search

In M-DFG, a data edge can transfer data from its source node to its target node. A path in M-DFG consisting of data edges can therefore transfer data from its start to its end. Given a M-DFG, one is able to statically check whether some data flows from a specified source node to a specified sink node by checking whether there exists a path in M-DFG linking them.

For each memory error (including the initial memory error), we add unintended RAW edge(s) as is described in §IV-D. Once they are added, we get all the unintended data-flow paths going through them. For each unintended data-flow, we check whether it starts from a sensitive information and ends with a kernel leaking sink (e.g., `copy_to_user`). If so, we find a potential *infoleak data-flow path*. In this work, we consider kernel pointers (function pointer, data pointer), keys and network & IPC messages (which may contain sensitive user data) as sensitive information (i.e., source nodes). For leaking sinks, we show the complete list in Table III. Given an M-DFG extended with unintended data-flow, we use a standard breath-first search (BFS) to find *all* paths between sources and sinks. For instance, in Figure 2, once the unintended RAW edge (red edge) is added, we can find a path from `bus` to `copy_to_user` going through the red edge.

When an infoleak path is found due to the adding of an unintended RAW edge resulting from a read primitive, we deem the infoleak path "read-induced infoleak" ($R_{\textbf{info}}$ for short); If it is due to the adding of an unintended RAW edge resulting from a write primitive, we deem the infoleak path "write-induced infoleak" ($W_{\textbf{info}}$ for short);

Not all infoleak data-flow paths in M-DFG will correspond to valid infoleak exploits. First, the feasibility of a data-flow relies on the feasibility of the corresponding control-flow path. For instance, consider a RAW edge, if there is no feasible control-flow path between a store node and the load node, then this RAW edge is actually infeasible. Therefore, an entire data-flow path is feasible iff there is at least on feasible control-flow path between every pair of data-flow nodes. However, because we omitted control-flow information when constructing M-DFG to make it scalable, we cannot reason about control-flow feasibility based on M-DFG. Second, due to the inherent imprecision of static points-to analysis, the RAW edges in M-DFG which are constructed based on static points-to analysis are not always feasible. As a result, the real data-flow may not follow the RAW edges in M-DFG.

To reduce false positives introduced by aforementioned imprecision in static analysis, we develop a dynamic verification process to verify each found data-flow path, before moving to the next iteration. For an infoleak data-flow path found in M-DFG, we verify two properties to ensure its feasibility: (1) its corresponding control-flow is feasible and (2) all RAW edges in the data-flow path are feasible. If so, the path is kept; otherwise it will be removed.

In this work, we first use symbolic execution to verify the feasibility of data-flow paths. Due to the scalability limitation of the symbolic execution engine we use (from [54]), if a data-flow path goes across multiple system calls (as in Figure 2), we divide it into multiple segments, each corresponds to a single system calls. These segments are connected through (intended or unintended) cross-syscall RAW edges (i.e., one syscall stores some data, and another syscall loads it). To verify the feasibility of the entire data-flow, we verify it segment by segment. If each segment is feasible and the RAW edges connecting them are feasible, then the entire data-flow is feasible. For instance, for the example in Figure 2, we need to run 3 symbolic executions for the 3 segments.

For each data-flow segment, we first find or construct a dynamic test case (e.g., from the syzbot PoC [13]–[15]) that can reach the starting point of the data-flow segment (e.g. first load node inside `ax25_setsockopt`, first load node inside `ax25_getsockopt` in Figure 2). Similar to SyzScope [54], we use the input to drive the execution to the first node of the data-flow segment, take a memory snapshot, and start symbolic execution from the memory snapshot. Based on the assumption of successful heap fengshui [49], we symbolize two types of memory in the memory snapshot: (1) attacker-controlled region (i.e., `copy_from_user()`'ed region (full control) and other object fields that are not pointers (limited control)) and (2) memory region containing the sensitive info (e.g., kernel pointer field) to be leaked. Note that we only consider these two types of memory within the UAF/OOB region or the memory that the previous data-flow segment writes to. The first type is used to explore execution paths that can be controlled by attackers. The second type is used to check if the leak indeed can happen and to track how the sensitive info will be transformed [26]. Note that, since we do not start the symbolic execution from the entry point of a syscall, we do not symbolize syscall arguments, and leave it for future work.

TABLE III: List of infoleak sinks.

```
unsigned long copy_to_user(void __user* to, const void* from, unsigned long n);
int nla_put(struct sk_buff* skb, int attrtype, int attrlen, const void* data);
int nla_put_nohdr(struct sk_buff *skb, int attrlen, const void* data);
int nla_put_64bit(struct sk_buff *skb, int attrtype, int attrlen, const void* data, int padattr);
void* nlmsg_data(const struct nlmsghdr* nlh); void* memcpy(void* dest, const void* src, size_t count);
void* nla_data(const structure nlattr *nla); void* memcpy(void* dest, const void* src, size_t count);
void* skb_put_data(struct sk_buff* skb, const void* data, unsigned int len);
void* skb_put(struct sk_buff* skb, unsigned int len); void* memcpy(void* dest, const void* src, size_t count);
int _printk(const char *fmt, ...)
```

To check for (1) control-flow feasibility, we perform a breadth-first search for feasible control-flow paths. If a feasible control-flow path that connects all data edges in the segment is found, we check for the second property. To check for (2) feasibility of RAW edges, we leverage symbolic execution's abilities to precisely reason about memory aliasing and to track data propagation. Specifically, after the symbolic executor find a feasible path, we check whether the data being stored is a symbolic formula of the symbolized sensitive information. If so, all RAW edges are feasible; otherwise at least one RAW edge is not feasible. For instance, for the `ax25_setsockopt` segment in Figure 2, we symbolize `&ax25_dev->values[N2]` as the sensitive info (because in previous segment `mon_bus_init()`, the same memory region contains a pointer) and symbolize other non-pointer fields as attacker-controlled, and check whether `&ax25->n2` is a formula of this symbolic value. Similarly, for the `ax25_getsockopt` segment, we symbolize `&ax25->n2` (because in previous segment the sensitive info will be written to the same memory) and check if the data being copied to the userspace is a formula of the symbolic value.

To check all segments as a whole are feasible, we record the control-flow path constraints for each feasible segment, then we put the constraints from all segments into constraints solver to check the feasibility. If there is no solutions, we deem it as infeasible. Right now we do not check the feasibility of cross-syscall RAW edges. For intended cross-syscall edges, we assume the point-to analysis [51] has accurately modeled it (e.g., kernel object retrieved from the file descriptor will be the same). For unintended cross-syscall edges, we assume it is feasible to perform heap fengshui [49] such that the store and load node will indeed access the same kernel object.

C. New Memory Error Search

To search for new memory errors, we look for pointers that can be controlled by the attacker. This can be translated into another searching for data-flow paths in M-DFG that starts with a node with attacker-controlled data, and ends with a pointer variable node. The pointer variable node is later used by a load or store node (i.e. connected to a load or store node through a pointer edge). If such data-flow path exists, the pointer variable can point to some unintended memory location, and a new memory error is created.

Similar to infoleak search, new memory error search also started after unintended data-flow edges are added. Start with nodes with attacker-controlled data, we propagate the data through data edges using flooding, until no new nodes are covered. Then we enumerate all data-flow paths that start from a node with attacker-controlled value and ends with a pointer variable node that is linked to a load / store node. Attacker-controlled value are marked in the graph in advance using prior knowledge. Specifically, we mark `copy_from_user` node as attacker-controlled; . Again, though we are searching for different types of data-flow paths (infoleak and attacker-controllable pointers), thanks to M-DFG, we can combine both searches into one round of breadth-first search.

We use the same dynamic verification process used in infoleak search to verify the feasibility of the data-flow path. Besides its feasibility, we also need to check on the value of the pointer variable, which determines the capability of the new memory error. Static analysis is one way to determine the value. One technique is to lazily redo the static points-to analysis along the path [43], updating the points-to set for all nodes along the path, including the controlled pointer at the end of the path, therefore reducing the overhead. This technique reflects the fact that the attacker-controlled value would propagate through the data-flow path to control the pointer variable. However, since static pointer analysis is often imprecise, the points-to set of the controlled pointer would be imprecise, thus the capability of the new memory error would not be precisely reason about in a static manner. In contrast, symbolic execution can precisely analyze the value of the controlled pointer. Therefore after knowing the controlled pointer in static analysis, we use symbolic execution to further determine the capability of the new error. A pointer is controlled when it contains symbolic value. During symbolic execution, we generate the feasible value for the controlled pointer. The feasible value of the controlled pointer represents its capability. If it is totally controllable (i.e. can take any value), it has arbitrary capability; If it is not totally controllable (i.e. can only take a limited range of value), then it is limited. For `copy_to_user()` and similar sinks (elastic object cases), in addition to the address argument, we additionally check whether its `len` argument is symbolic and extracts its constraints to understand how many bytes can be copied to userspace. Currently, we limit the max number of bytes to leak to 4KB.

When an new memory error is derived due to the adding of an unintended RAW edge resulting from a read primitive, we deem the new memory error "read-induced new error" ($R_{\textbf{nerr}}$ for short). This means that a read error creates a new error; If it is due to the adding of an unintended RAW edge resulting from a write primitive, we deem the infoleak path "write-induced new error" ($W_{\textbf{nerr}}$ for short). This means that a write error creates a new error;

D. Iterative Search Algorithm

The infoleak search and new memory error search are combined into our iterative algorithm. The high-level workflow

has been shown in Figure 1: in each iteration, we first extend M-DFG using newly added memory errors (the initial memory error is considered as new for the first iteration), then we search for infoleak paths and new memory errors; newly found memory errors are then fed to the next iteration. This process repeat until we have reached a threshold, or no new infoleak and memory errors are found. In Algorithm 1, we give a detailed algorithm.

Using our iterative search algorithm on top of M-DFG, we are able to do a systematic search for infoleak exploits. The iterative search algorithm naturally categorizes different infoleak strategies, and thus the infoleak search space is naturally divided by the algorithm. An infoleak strategy is defined by *how* and *in which iteration* the infoleak path is output in the iterative algorithm. Here are several examples: If the initial memory error is a read error, and the first iteration of the algorithm produces the infoleak, the strategy is considered $R_{\textbf{info}}$; If the initial memory error is a write error, and the first iteration of the algorithm produces the infoleak, the strategy is considered $W_{\textbf{info}}$; If the initial memory error is a write error, and the first iteration of the algorithm finds a new read error (i.e. it is a write-induced error, $W_{\textbf{nerr}}$), and a later iteration of the algorithm finds an infoleak using the read error, then the strategy is considered $W_{\textbf{nerr}} + R_{\textbf{info}}$. So on and so forth.

E. Exploitability Verification

Once K-LEAK outputs a potential infoleak path, where each data-flow segment has been verified by symbolic execution, we perform an additional manual analysis to further verify its exploitability. We first construct a cross-syscall test case (i.e., syscall sequences) that connects all data-flow segments. Then we use this test case to confirm the control-flow and data-flow feasibility of the infoleak path. The only assumption we make in this step is the success of heap fengshui. That is, we simulate an illegal read-after-write data-flow by using GDB to "copy" stored value to the read value. For example, in Figure 2, the leaking data-flow path starts with an UAF read that reads a kernel pointer from the victim object `mbus`. In a real exploit, we need to perform heap fengshui to reallocate an `mbus` object in the freed heap slot previoiusly belongs to an `ax25_dev` object. However, because automated heap fengshui in the kernel is still an open problem, we simulate a successful fengshui by using GDB to pass the kernel pointer `mbus->u_bus` to the load instruction (i.e., `tmp = mbus->u_bus`). If we indeed can observe the sensitive information being leaked by the test case, we deem the corresponding memory error as exploitable.

VI. Implementation

We use static analysis and symbolic execution to implement a prototype of the framework that runs our iterative algorithm to generate infoleak exploits. In this section, we will describe the implementation details of the prototype. The prototype takes the source code of the kernel and a KASAN bug report as input, and outputs potential infoleak exploit(s).

A. Input: Memory Error from syzbot

We choose the memory errors found by syzbot [23] as the input to our prototype. Syzbot is a continuous fuzzer that fuzzes the Linux kernel and releases a bug report and a bug reproducer (i.e. syscall input sequence) for each found bug. The underlying fuzzer used by syzbot is Syzkaller. With the fuzzed kernel instrumented with different types of sanitizers, syzbot is able to detect different types of bugs during fuzzing and accordingly produce a bug report and a reproducer for each found bug. Among the bugs found by syzbot, we focus on heap use-after-free and heap out-of-bound memory errors, two most common types of memory errors in real world. The reports are generated by KASAN sanitizer. An example report is shown in Listing 1. Syzbot will stop the kernel execution when the first memory error is detected. Although the bug may trigger multiple memory errors, the report from syzbot only contains the first memory error, rendering the report being incomplete. Syzscope [54] is able to obtain a more complete set of memory errors. With the help of Syzscope, after the first memory error is triggered, later errors can still be analyzed as the kernel continues executing. Therefore, we use Syzscope to get the set of memory errors (in the form of UAF/OOB capabilities) for a single bug found by syzbot. They will be considered as the set of *initial* memory errors as input to K-LEAK.

B. M-DFG Builder

Besides the initial memory errors, the kernel source code is another piece of input to the prototype. Using clang compiler, it is first compiled into LLVM IR, which subsequent static analysis and the M-DFG data structure are based on. Our analyses are implemented using the C++ LLVM Compiler infrastructure. To deal with the symbol renaming issues in the clang compiler, we process individual IR files instead of linking them.

Based on the compiled IR files, we use SUTURE to do a cross-syscall points-to analysis and collect the intended points-to information. Since SUTURE is a summary-based points-to analysis that summarize the behaviors each syscall entry, we need to specify the set of syscall entries to be summarized. To reduce the overhead, we only run the analysis on a limited number of syscall entries: (1) the syscall entry that syzbot input invokes to trigger the memory error (e.g. `ax25_setsockopt` in Listing 2); (2) the syscall entries in the same kernel module as the memory-error-triggering entry (e.g. `ax25_getsockopt` in Listing 2). (3) the syscall entries that are analyzed by published work [13]–[15]. The published works give concrete test cases, so we collect them into a database. Since the number of them are limited, we additionally craft some and add them to the database. We also use such database to collect the dynamic indirect call graph edges, and use them in cross-syscall points-to analysis mentioned earlier.

We build and store M-DFG in C++ program. To store M-DFG representation, we use the graph library [1] to maintain the vertices and edges. Also we use multiindex library [2] to maintain the points-to information, as well as other auxiliary information related to the vertices or edges, which speeds up data retrieval using index.

C. Graph Searcher

When implementing our iterative algorithm, we limit the number of iteration to 2. We found the results converged within 2 iterations, given our analysis scope (i.e. syscalls and objects).

Algorithm 1 Iterative Search Algorithm

```
1: Input Intended M-DFG, the capability of an initial memory error ic
2: Output infoleak paths leakPaths
3: leakPaths := ∅
4: worklist := ∅
5: Extend M-DFG with a set of RAW edges E according to ic
6: ENQUEUE(worklist, E)
7: while worklist ≠ ∅ do
8:     e := DEQUEUE(worklist)
9:     paths := PATHS_THROUGH_EDGE(e)
10:    for each path ∈ paths do
11:        if IS_INFOLEAK_PATH(path) then
12:            if VERIFY(path) then
13:                leakPaths := leakPaths ∪ {path}
14:            end if
15:        end if
16:        if IS_NEWERROR_PATH(path) then
17:            if VERIFY(path) then
18:                nc := GET_CAP(path)
19:                Extend M-DFG with E′ according to nc
20:                ENQUEUE(worklist, E′)
21:            end if
22:        end if
23:    end for
24: end while
```

When searching for data flow paths in M-DFG, we use BFS to search for infoleak paths and new memory errors, and we limit the search depth to avoid excessively long data-flow paths. After interesting nodes are visited (e.g. infoleak sink, pointer variable connecting to a load or store node), we use a backward DFS to construct the data-flow path of interest (i.e. infoleak path or path creating new memory error).

In symbolic execution, we use the same framework as [54], which is a symbolic executor based on Angr [40]. It takes a kernel snapshot as input, and runs dynamic symbolic execution on it. It can explore about 2,000 basic blocks within the 4-hour time budget. Its concretization strategy is similar to Angr's default one (e.g., a symbolic index will be concretized), which can lead to false negatives during feasibility verification.

VII. Evaluation

In this section, we evaluation our prototype of K-LEAK to answer the following research questions:

- **RQ1**: how effective K-LEAK is on generating Linux kernel infoleak exploit?
- **RQ2**: does modeling cross-syscall data-flow allow K-LEAK to generate more exploits?
- **RQ3**: do new memory errors discovered during multi-iteration search allow K-LEAK to generate more exploits?
- **RQ4**: is the graph-based search fast enough to handle Linux kernel?
- **RQ5**: how effective is K-LEAK compared to manual analysis?

A. Evaluation Data Set

We select from syzbot 250 real-world fuzzer-exposed memory bugs (found before 2022 and already fixed) for upstream kernel to evaluate the ability of K-LEAK in generating infoleak exploits. We use keywords "use-after-free" and "out-of-bound" in the report titles to search for these bugs (e.g., the title of Listing 1 contains keyword "use-after-free"). Within the search results, we then select those that we can reliably trigger (i.e., reproduce the KASAN warning message using the bug reproducer). The distribution of the 250 bugs are: 180 heap UAF/OOB read, 70 heap UAF/OOB write. For comparison with ELOISE [14], we confirm that all 10 bugs from syzbot used in ELOISE are included in our evaluation dataset.

B. Overall Results

The overall evaluation results are shown in Table IV. Due to page limit, this table only contains bugs that have verified infoleak paths, including the 10 bugs also used in ELOISE [14]. Each row in the table contains the results for one memory bug. The *ID* column shows the ID for each syzbot bug. Those IDs followed by "(EL)" are also those used in [14]. The *Type* column shows the type of the bug (i.e., UAF or OOB). The *Initial Memory Errors* column shows the initial set of memory errors of by each bug. "R" represents read error, and "W" represents write error. For example, "14 R 3 W" means the initial set of memory errors include 14 read errors and 3 write errors.

Since we use an iterative algorithm to search for infoleak, we also show the intermediate results produced during the iterative algorithm. The *1st Iteration* and *2nd Iteration* columns shows the results from corresponding round when running the iterative algorithm (Algorithm 1). It shows the number of infoleak paths (denoted as "infoleak") and the number of

TABLE IV: Exploit generation results. Each row in the table contains the results for one memory bug. The **ID** column shows the syzbot bug ID. IDs marked with "(**EL**)" are those also evaluated in ELOISE [14]. **Type** column shows the type of the bug. **Initial Memory Errors** column shows the initial set of memory errors (i.e., read and write operations using the invalid pointer). **R** represents read error, and **W** represents write error. **1st Iteration** and **2nd Iteration** columns shows the results when running the iterative algorithms. Results before the "/" are from the Static Analysis, and after are the results verified by Symbolic Execution: **infoleak** means number of infoleak paths (the corresponding exploit strategy is in the parentheses), and **memerr** means number of new memory errors (the type of created new memory error is in the parentheses). **Infoleak Paths** column summarizes the overall numbers of infoleak paths for the case. **Strategy Summary** column shows the infoleak strategy used to exploit each syzbot case. **Cross syscall** column shows whether any infoleak data-flow crosses more than one syscalls (either intended or unintended), **T** means crosses.

ID	Type	Initial Memory Errors	1st Iteration Static Analysis / SE-Verified	2nd Iteration Static Analysis / SE-Verified	Infoleak Paths	Strategy Summary	Cross-Syscall
01b3316	UAF	1 R	1 infoleak ($R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{info}}$)	- / -	1	$R_{\textbf{info}}$	**T**
059cee5	UAF	1 R	1 infoleak ($R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{info}}$)	- / -	1	$R_{\textbf{info}}$	
1c07845	UAF	11 R	7 infoleak ($R_{\textbf{info}}$) / 3 infoleak ($R_{\textbf{info}}$)	- / -	3	$R_{\textbf{info}}$	**T**
af50cf6	UAF	1 R	1 infoleak ($R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{info}}$)	- / -	1	$R_{\textbf{info}}$	
cd4d85b	UAF	1 R	1 infoleak ($R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{info}}$)	- / -	1	$R_{\textbf{info}}$	
5aae242	UAF	1 R	2 infoleak ($R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{info}}$)	- / -	1	$R_{\textbf{info}}$	
148d2f1	UAF	5 R	1 memerr (R) / 1 memerr (R) 1 infoleak ($R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{info}}$)	1 infoleak ($R_{\textbf{nerr}} + R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{nerr}} + R_{\textbf{info}}$)	2	$R_{\textbf{info}}$ $R_{\textbf{nerr}} + R_{\textbf{info}}$	
0655ccf	UAF	4 R 1 W	1 memerr (R) / 1 memerr (R)	1 infoleak ($R_{\textbf{nerr}} + R_{\textbf{info}}$) / 1 infoleak ($R_{\textbf{nerr}} + R_{\textbf{info}}$)	1	$R_{\textbf{nerr}} + R_{\textbf{info}}$	
1d22a2c	UAF	2 W	4 infoleak ($W_{\textbf{info}}$) / 4 infoleak ($W_{\textbf{info}}$)	- / -	4	$W_{\textbf{info}}$	**T**
2d4684c	UAF	7 R 2 W	4 infoleak ($W_{\textbf{info}}$) / 4 infoleak ($W_{\textbf{info}}$)	- / -	4	$W_{\textbf{info}}$	**T**
64b6c15	UAF	1 W	2 infoleak ($W_{\textbf{info}}$) / 2 infoleak ($W_{\textbf{info}}$)	- / -	2	$W_{\textbf{info}}$	**T**
9ea5654	OOB	4 R 5 W	4 infoleak ($W_{\textbf{info}}$) / 4 infoleak ($W_{\textbf{info}}$)	- / -	4	$W_{\textbf{info}}$	**T**
45591ae	UAF	2 W	4 infoleak ($W_{\textbf{info}}$) / 4 infoleak ($W_{\textbf{info}}$)	- / -	4	$W_{\textbf{info}}$	**T**
8670f2d	UAF	1 W	1 infoleak ($W_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{info}}$)	- / -	1	$W_{\textbf{info}}$	**T**
e2554a4	UAF	3 R 2 W	4 infoleak ($W_{\textbf{info}}$) / 4 infoleak ($W_{\textbf{info}}$)	- / -	4	$W_{\textbf{info}}$	**T**
e9a87c1	OOB	4 R 6 W	4 infoleak ($W_{\textbf{info}}$) / 4 infoleak ($W_{\textbf{info}}$)	- / -	4	$W_{\textbf{info}}$	**T**
1379 (**EL**)	UAF	14 R 3 W	10 memerr (R) / 1 memerr (R)	10 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$)	1	$W_{\textbf{nerr}} + R_{\textbf{info}}$	
3d67 (**EL**)	OOB	1 W	- / -	- / -	0	-	
422a (**EL**)	OOB	1 W	- / -	- / -	0	-	
5bb0 (**EL**)	UAF	1 W	1 infoleak ($W_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{info}}$) 1 memerr (R) / 1 memerr (R)	1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$)	2	$W_{\textbf{info}}$ $W_{\textbf{nerr}} + R_{\textbf{info}}$	**T**
6a6f (**EL**)	UAF	1 W	3 memerr (R) / 1 memerr (R)	3 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$)	1	$W_{\textbf{nerr}} + R_{\textbf{info}}$	
a84d (**EL**)	OOB	1 W	1 memerr (R) / 1 memerr (R)	1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$)	1	$W_{\textbf{nerr}} + R_{\textbf{info}}$	
bf96 (**EL**)	UAF	1 W	- / -	- / -	0	-	
e4be (**EL**)	OOB	3 W	6 memerr (R) / 0 memerr (R)	6 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$) / 0 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$)	0	-	
e928 (**EL**)	UAF	1 W	1 memerr (R) / 0 memerr (R)	1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$) / 0 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$)	0	-	
ebeb (**EL**)	UAF	1 W	1 infoleak ($W_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{info}}$) 1 memerr (R) / 1 memerr (R)	1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$) / 1 infoleak ($W_{\textbf{nerr}} + R_{\textbf{info}}$)	2	$W_{\textbf{info}}$ $W_{\textbf{nerr}} + R_{\textbf{info}}$	**T**

paths that create new memory errors (denoted as "memerr"). Numbers go before the "/" are the static analysis results from the iteration, and numbers after the "/" are the results verified by symbolic execution. In the example of `148d2f1`, in the first iteration, the static analysis finds 1 data-flow path resulting in a read primitive (marked as "R"), and the symbolic execution verifies the primitive; the static analysis also finds 1 infoleak path, and the symbolic execution verifies the path.

The *Infoleak Paths* column shows the overall number of infoleak data flow paths for the case. The *Strategy* column shows the infoleak strategies (see §V-D) used to exploit each syzbot case. The *Cross syscall* column shows whether any of the verified infoleak paths crosses more than one syscalls (in an intended or unintended manner).

To answer **RQ1**, we look at the number of syzbot cases that K-LEAK can find verified infoleak paths. Among the 250 syzbot bugs, K-LEAK is able to find infoleak paths for 40 cases. all 40 cases can leak full/transformed kernel pointers. Table IV lists 21 cases that are *promising* (see definition in §V-E) for end-to-end exploit. All the 10 infoleak cases from ELOISE [14] can also be found by K-LEAK statically. We are able to generate diverse exploits in terms of strategy and the number of infoleak paths. There are a total of four kinds of infoleak strategies used: $R_{\textbf{info}}$, $R_{\textbf{newerr}} + R_{\textbf{info}}$, $W_{\textbf{info}}$, $W_{\textbf{newerr}} + R_{\textbf{info}}$. $R_{\textbf{info}}$, $R_{\textbf{newerr}} + R_{\textbf{info}}$ and $W_{\textbf{info}}$ cannot be handled by [14]. Some cases (e.g. `148d2f1`) can be exploited using more than one strategy. Many cases have more than 1 infoleak paths. Based on these results, we think the answer to RQ1 is *yes*.

K-LEAK can fail to generate exploits for the remaining cases due to the following reasons. (1) In our current prototype, we leverage SyzScope [54] to reason about the capability of the initial memory error, since SyzScope uses a kernel context prepared by the PoC input, the extracted capability may be more restrictive than it could be. (2) When extending M-DFG with unintended data-flow, we only consider a relatively small number of kernel objects (see §IV-D). A combination of (1) and/or (2) can result in K-LEAK failing to find unintended

data-flow that can read sensitive data or introduce new memory error. (3) When verifying an infoleak path or new memory error with symbolic execution, we use a testcase to prepare the kernel context; Thus the syscall arguments are concrete values from the test cases. This could make the context over-constrained thus prevent SE from finding a feasible path. On average for each case, 2,400 bytes of memory are symbolized (other memory is concrete). Finally, (4) when constructing M-DFG, we only consider syscalls that can be covered by testcases, so K-LEAK can miss data-flow through syscalls it does not know how to invoke.

To answer **RQ2**, we look at the number of infoleak paths that cross syscalls (in an intended or an unintended manner). The infoleak paths of 2 cases cross syscalls in an intended manner, and those of 10 cases cross syscalls in an unintended manner. We consider that an infoleak exploit crosses syscall if it crosses syscall after the buggy syscall. For example, in Figure 2, the red that links `mon_bus_init()` to `ax25_setsockopt()` does not count; the edge that links `ax25_setsockopt()` to `ax25_getsockopt()` does count. When we count the number of infoleak paths, we also apply the same criterion.

To answer **RQ3**, we look at the number of infoleak paths that are output by the 2nd iteration. This indicates that the infoleak paths result from new memory errors. The number is 9. Excluding those that use the strategy of elastic objects [14], the number is 3. They are exploited by strategy $R_{\textbf{nerr}} + R_{\textbf{info}}$.

To answer **RQ4**, we look at the size of M-DFG and the time it takes to statically search M-DFG for potential infoleak paths and new memory errors. When doing the evaluation for each syzbot memory bug, we log the size of M-DFG and the search time. For the size of M-DFG, the average context-sensitive CFG size being analyzed for each bug is around 500,000 basic blocks. The M-DFG built for each bug contains roughly 1,000,000 nodes and edges, respectively, on average. For the static search time, the shortest time is 7ms; The longest time is 81ms; and the median time 51ms. The short static search time is short, because the path in M-DFG is usually short, resulting in the termination of search. During the graph search, the average length of each data flow is 12 nodes. Also, the infoleak paths found are usually short, typically under 10 nodes. But their corresponding CFG paths can span up to 100 basic blocks. To conclude, our graph-based search approach is scalable to handle large programs like the Linux kernel.

To answer **RQ5**, we evaluate K-LEAK using public exploits to determine if K-LEAK can discover infoleak opportunities from the same vulnerabilities. This will illustrate the effectiveness of K-LEAK in comparison with manual analysis performed by human experts. To this end, We include 11 additional CVEs (found between 2020 and 2022) that have publicly available exploits with an infoleak based on memory errors. We compared the exploits generated by K-LEAK and public exploits by human experts. The results are shown in Table V. K-LEAK is able to detect infoleaks for 7 out of the 11 CVEs. For the remaining 4 CVEs, there are 3 reasons why the infoleaks in public exploits can not be found: (1) The exploitation process involves creating illegal free primitives, which is currently not modeled in K-LEAK. In the iterative algorithm, we search for new memory read or write errors, which results from a controlled pointer being used to read or write. However, K-LEAK does not model the scenario where the controlled pointer is used in a free operation, e.g., kfree(). This can add additional memory error exploitation search space — 3 public exploits make use of it to illegally free some kernel objects. We believe in the future we can additionally model this in M-DFG by introducing a special free node. (2) The exploit utilizes stack memory error, whereas K-LEAK only models heap memory errors — this happens in the 4th public exploit. (3) The exploit utilizes infoleak through control flow, whereas K-LEAK only considers infoleak through data-flow — this happens in the same 4th exploit. Here is a simplified version of it: `if(controlledVal==sensitiveInfo)print("success");`. In this example, attackers can change `controlledVal` until he sees a success message in order to leak `sensitiveInfo`. Also note that among the syzbot dataset only 1 has public exploit and K-LEAK did successfully find the infoleak.

TABLE V: Evaluation results using CVEs with public exploits.

ID	Strategy in public exploits	K-LEAK found strategy
CVE-2020-8835	$R_{\textbf{info}}$	$R_{\textbf{info}}$
CVE-2021-22555	Creating free primitive	-
CVE-2021-42008	$W_{\textbf{nerr}} + R_{\textbf{info}}$	$W_{\textbf{nerr}} + R_{\textbf{info}}$
CVE-2021-43267	$W_{\textbf{nerr}} + R_{\textbf{info}}$	$W_{\textbf{nerr}} + R_{\textbf{info}}$
CVE-2022-0185	$W_{\textbf{nerr}} + R_{\textbf{info}}$	$W_{\textbf{nerr}} + R_{\textbf{info}}$
CVE-2022-0995	Creating free primitive	-
CVE-2022-1015	Stack & control-flow infoleak	-
CVE-2022-25636	Creating free primitive	-
CVE-2022-2639	$W_{\textbf{nerr}} + R_{\textbf{info}}$	$W_{\textbf{nerr}} + R_{\textbf{info}}$
CVE-2022-27666	$W_{\textbf{nerr}} + R_{\textbf{info}}$	$W_{\textbf{nerr}} + R_{\textbf{info}}$
CVE-2022-32250	$W_{\textbf{info}}$	$W_{\textbf{info}}$

End-to-end Exploits. With the ability of K-LEAK in helping infoleak generation, we also develop end-to-end exploits for 7 bugs: `1d22a2c`, `9ea5654`, `1379b6b`, `059cee5`, `2d4684c`, `b8febdb`, `e9a87c1`. K-LEAK outputs the infoleak data-flow path as well as the correspondent control-flow path. Since K-LEAK is not end-to-end, the main task remains is to manually craft the exploit program to achieve heap manipulation.

C. Case Study

1) W strategy and unintended cross-syscall data flow: We study the memory error reported by syzbot with case id 9ea5654 [8]. In this case, the set of initial memory errors consists of 4 slab out-of-bound read errors and 5 slab out-of-bound write errors (see Table IV). Among them three write errors are of interest: one write writes a heap pointer (Listing 3 line 7), and the other two writes a function pointer to slab out-of-bound memory belonging to `kmalloc-192` (Listing 3 line 8 and line 9).

In the first iteration of the algorithm, since the capability of the memory errors is in `kmalloc-192`, K-LEAK pair the errors with `kmalloc-192` memory object `struct user_key_payload`. Then unintended RAW edges are added to connect the memory error store nodes to a load node (`copy_to_user` node, to be precise) that loads the memory of object `struct user_key_payload` to user space. Therefore, there will exist three data-flow paths that start with a sensitive information and end with leaking sink. In this case, with the leakage of kernel function pointer, KASLR can be bypassed. We have developed an end-to-end exploit for this case.

```
// syzcall 1
int input_ff_create(struct input_dev *dev, unsigned int
↪  max_effects) {
  struct ff_device *ff;
// ...
  ff = kzalloc(ff_dev_size, GFP_KERNEL);
// ...
  dev->ff = ff; // OOB Write: heap pointer
  dev->flush = input_ff_flush; // OOB Write: function ptr
  dev->event = input_ff_event; // OOB Write: function ptr
}

// syzcall 2
long user_read(const struct key *key, char __user *buffer,
↪  size_t buflen) {
  const struct user_key_payload *upayload;
  upayload = user_key_payload_locked(key);
  buflen = upayload->datalen;
  if (copy_to_user(buffer, upayload->data, buflen) != 0)
    ret = -EFAULT;
}
```

Listing 3: $W_{\textbf{info}}$ strategy; Unintended cross-syscall data flow

```
long drm_ioctl(struct file *filp,
               unsigned int cmd, unsigned long arg) {
  retcode = drm_ioctl_kernel(filp, func, kdata,
↪  ioctl->flags);
  if (copy_to_user((void __user *)arg, kdata, out_size) != 0)
    retcode = -EFAULT;
}

int drm_getunique(struct drm_device *dev, void *data,
                  struct drm_file *file_priv) {
  struct drm_unique *u = data;
  struct drm_master *master = file_priv->master;
    if (copy_to_user(u->unique, master->unique,
↪  master->unique_len)) {
      mutex_unlock(&master->dev->master_mutex);
      return -EFAULT;
    }
  u->unique_len = master->unique_len;
}
```

Listing 4: Exploited by two strategies: $R_{\textbf{info}}$ and $R_{\textbf{nerr}} + R_{\textbf{info}}$

2) Exploited using more than one strategy: We also study the memory error reported by syzbot with case id 148d2f1. In `drm_getunique` function in Listing 4, `struct drm_master *master` is a dangling pointer. Therefore, `master->unique`, `master->unique_len` are two pointer controllable by attackers. They create a new memory read error at `copy_to_user`. According to our definition, this is exploited using $R_{\textbf{nerr}} + R_{\textbf{info}}$ strategy.

Also, `u->unique_len = master->unique_len;` is a memory read error among the initial set of memory errors. In the first iteration, we find an infoleak sink at another `copy_to_user` in `drm_ioctl`. According to our definition, this is exploited using $R_{\textbf{info}}$ strategy.

D. Threat to validity

Even though we successfully developed 7 end-to-end exploits of infoleaks, we discuss the potential threats to validity of our general results here. Generally speaking, there are three types of issues that can affect the end-to-end exploitability.

First, we assume heap fengshui or heap manipulation is always successful. As mentioned in §V-E, our current solution bypasses the step of heap manipulation by using GDB to forcefully change the value of key objects across syscalls. In reality, heap manipulation is challenging and can affect the stability and success rate of exploits. Recently, there are several approaches on automating heap manipulation strategies [21], [27], [32], [47], [50]. Yet there are no open-sourced tools for Linux kernel yet. Unfortunately, it is generally considered much more difficult to handle race condition bugs which can create uncertain memory layout and significant noises [49]. Nevertheless, we consider this an orthogonal step where heap manipulation strategies can be fine-tuned to eventually improve the infoleak success rate.

Second, kernel liveness issue can affect the end-to-end exploitability. This happens when the attacker-controlled data may crash the kernel. For example, the attacker-sprayed NULL bytes may be used to as a pointer and dereferenced, which crashes the kernel. From the 40 cases that we deemed potentially exploitable, we find that 15 of them involve pointer dereferences on attacker-controlled data. Even though this issue may potentially be overcome by carefully crafting the attacker-controlled data, we conservatively consider such cases more difficult to exploit. Note that symbolic execution can detect such issues, but a fully automated solution to address them is left to future work.

Third, we observe that there can be *constraints imposed over sensitive info* such that it makes the infoleak useless. For example, when the attacker-controlled data is constrained to some other sensitive information (e.g., `controlledData==kernel_ptr`), it will require the attacker to guess such info ahead of time before being able to leak what we need (a chicken-and-egg problem). Through our analysis, we find 11 out of the 40 cases that fall under this category. Depending on the nature of the constraints, such bugs may or may not be exploitable. To be conservative, we consider such cases to be unlikely exploitable. Note that this issue can be detected by symbolic execution automatically. However, we leave a complete solution to automatically discern what constraints are still exploitable to future work.

Overall, we conservatively define *promising* infoleaks to be those that are free from the second and third issue, as they can potentially invalidate the exploit completely. This leaves 21 out of the 40 cases considered promising (note that some bugs experience both the second and third issue).

VIII. Discussion & Future Work

Comparison with AlphaEXP. AlphaEXP [46] is a reasoning framework for memory-error-based kernel exploit. Compared with our work, AlphaEXP has a larger scope as it reasons about more types of kernel exploits (e.g. corrupting a pointer to achieve control-flow-hijacking). However, the main focus of AlphaEXP is to find objects useful for exploits, especially for upgrading a weak primitive into a strong primitive. Infoleak is not the focus of AlphaEXP and it misses precise modeling and reasoning of data-flow for generating infoleak exploit.

Dynamic Traces. In this work, when building M-DFG for syscall entries, we use those confirmed (e.g., by us or published work) to have dynamic test cases. This is another hurdle to automate infoleaks from end to end. Recent exploit generation

papers [14], [46] have not yet open-sourced or just a subset of their verified dynamic traces. K-LEAK will benefit from recent work that automates the recovery of syscall interfaces [12], [18], [25] and performs better kernel fuzzing [45], [52], which can facilitate the generation of more dynamic traces. Also, our dynamic symbolic execution requires a dynamic trace to begin with. We can avoid this issue by using a symbolic execution starting at the beginning of the syscall.

Capability Reasoning. When reasoning about the capability of OOB bug from syzbot, we rely on KASAN report. However, the ability of KASAN to reason about OOB has limitations [13]. In the future, we can utilize prior work such as KOOBE [13] to better reason about its capability. Doing so could increase the chance of successful exploit.

Future Use Cases of M-DFG. The benefit of M-DFG is not limited to infoleak exploit generation. We believe M-DFG can also be used in general memory-error-based exploit generation. As modeled in the conceptual framework for exploiting memory errors [44] and in the recent knowledge graph [46], exploiting a Linux kernel memory error can usually be partitioned into two stages: in the first stage, attackers try to create as many unintended data flows as possible, including introducing new memory errors; in the second stage, they utilizes the unintended data flows to achieve the attacking goals (e.g., arbitrary code execution, and privilege escalation). Although the second stage will be different, M-DFG can be used as a general infrastructure to search for exploitable Linux kernel data flows. For example, when exploiting an initial kernel use-after-free vulnerability to achieve arbitrary code execution, attackers can first use M-DFG search for unintended data flows that cause invalid writes; then use the invalid writes to overwrite a function pointer to hijack the control-flow.

IX. Related Work

Sleak [26] is a related work on userspace infoleak. It reasons about how internal information can be leaked through output functions, and reconstructs the original info. Both Sleak and K-LEAK use static analysis and symbolic execution to reason about the feasibility of a leak, and both can reconstruct the original info if the leaked data has been transformed. The main difference, however, is that Sleak only reason about infoleak through *legitimate* data flows; while K-LEAK aims to find *memory-error-based* infoleak. In particular, K-LEAK (1) models *illegal* data flows introduced by memory errors, and (2) reasons about how existing memory errors can introduce new memory errors, and (3) reasons about cross-entry (cross-syscall) data flows.

Kernel Information Leak. Kernel information leak can be reached by multiple attack surface. ELOISE [14] uses static taint analysis to locate the kernel objects that can transit kernel data to userspace. It abuses the buffer size to create out-of-bound read in kernel space. Cho's work [16] solely focuses on information leaks from uninitialized stack variables, it brought a new leaking surface and their tool was proven efficient for such leaks. Some studies [24], [30] proposed side-channel based approach to bypass SMAP or KASLR. They achieve that by utilizing the time difference between CPU cache delay.

Exploitability Assessment. Assessments of bug exploitability contribute to kernel security. SyzScope [54] inspects common kernel bugs and reveals their high-risk primitives. High-risk primitives like use-after-free write, and out-of-bounds write, are necessities for most kernel exploits [3]–[5] Some automatic exploit generation tools target such primitives and try to generate exploit without human supervision. KOOBE [13] takes out-of-bounds bugs, applying fuzzing and symbolic execution to explore the hidden exploitability and craft the final exploits. FUZE [48] uses symbolic execution to search for more capability within the use-after-free memory. Orthogonally, SyzBridge [53] investigates the gap of bug exploitability between upstream kernel and downstream kernel.

Exploit Knowledge Graph. Recent works also make use of knowledge graph in aiding Linux kernel exploit generation. The knowledge graph built by AlphaEXP [46] can help reason about what objects are security sensitive in terms of creating new memory corruptions by corrupting the objects. However, such knowledge graph doesn't fully reason about the data-flow in the OS kernel, since it only maintains high-level relations among different kernel entities (e.g. functions, objects).

X. Conclusion

In this paper, we propose a graph-based framework K-LEAK that automate the exploit generation memory-error-based infoleak exploits for the Linux kernel. We implement the framework that utilizes static analysis and dynamic verification to generate infoleak exploits for kernel memory errors. Evaluation our implementation with 250 fuzzer-exposed memory errors from syzbot demonstrates that our approach can generate diverse infoleak exploits, including new strategies that are not explored in previous work. We open source our implementation at https://github.com/seclab-ucr/K-LEAK to facilitate future research.

Acknowledgment

The authors would like to thank the reviewers for their valuable feedback during the revision process. This work is supported by the National Science Foundation under Grant #2155213, #1953933, #1652954 and #2046026.

References

[1] "Boost Graph," https://www.boost.org/doc/libs/1_81_0/libs/graph/doc/index.html.

[2] "Boost Multi Index," https://www.boost.org/doc/libs/1_81_0/libs/multi_index/doc/index.html.

[3] "CVE-2022-0185," https://www.willsroot.io/2022/01/cve-2022-0185.html.

[4] "CVE-2022-1786," https://blog.kylebot.net/2022/10/16/CVE-2022-1786/.

[5] "CVE-2022-27666," https://etenal.me/archives/1825.

[6] "Kernel control flow integrity," https://source.android.com/docs/security/test/kcfi.

[7] "syzbot case 1c07845," https://syzkaller.appspot.com/bug?id=1c07845d565d5e670218fde04599322aa81a3bf8.

[8] "syzbot case 9ea5654," https://syzkaller.appspot.com/bug?id=9ea5654403357926bb929ff049fd64df3b0d260e.

[9] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*, 2020.

[10] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.

[11] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 147–160.

[12] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, 2021.

[13] W. Chen, X. Zou, G. Li, and Z. Qian, "Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 1093–1110.

[14] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1165–1184.

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

[16] H. Cho, J. Park, J. Kang, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 1–1.

[17] J. Corbet, "Supervisor mode access prevention," https://lwn.net/Articles/517475/.

[18] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3133956.3134069

[19] J. Edge, "Kernel address space layout randomization," https://lwn.net/Articles/569635/.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[21] J. Gennissen and D. O'Keeffe, "Hack the heap: Heap layout manipulation made easy," in *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022, pp. 289–300.

[22] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, 2020.

[23] Google, "syzbot," https://syzkaller.appspot.com/upstream/, 2023.

[24] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 368–379.

[25] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *IEEE Symposium on Security and Privacy*, 2023.

[26] C. Hauser, J. Menon, Y. Shoshitaishvili, R. Wang, G. Vigna, and C. Kruegel, "Sleak: Automating address space layout derandomization," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 190–202.

[27] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *USENIX Security Symposium*, 2018.

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

[29] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[30] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 191–205.

[31] A. Konovalov, "Exploiting the linux kernel via packet sockets," https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[32] R. Li, B. Zhang, J. Chen, W. Lin, C. Feng, and C. Tang, "Towards automatic and precise heap layout manipulation for general-purpose programs," in *Network and Distributed System Security Symposium (NDSS)*, 2023.

[33] K. Lu, "Securing software systems by preventing information leaks." Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2017.

[34] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang, "Checking more and alerting less: detecting privacy leakages via enhanced data-flow analysis and peer voting." in *NDSS*, 2015.

[35] A. Møller and M. I. Schwartzbach, "Static program analysis," *Notes. Feb*, 2012.

[36] MosheKol, "Racing against the lock: Exploiting spinlock uaf in the android kernel," https://0xkol.github.io/assets/files/Racing_Against_the_Lock__Exploiting_Spinlock_UAF_in_the_Android_Kernel.pdf.

[37] A. Orailoglu and D. D. Gajski, "Flow graph representation," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, 1986, pp. 503–509.

[38] A. Popov, "Four bytes of power: Exploiting cve-2021-26708 in the linux kernel," https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.

[39] F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012.

[40] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[41] W. Song, "Bypassing smep+pti+smap," https://github.com/pr0cf5/kernel-exploit-practice/blob/master/bypass-smap/README.md.

[42] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 32–41.

[43] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[44] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[45] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh, "SyzVegas: Beating kernel fuzzing odds with reinforcement learning," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[46] R. Wang, K. Chen, C. Zhang, Z. Pan, Q. Li, S. Qin, S. Xu, M. Zhang, and Y. Li, "Alphaexp: An expert system for identifying security-sensitive kernel objects," in *USENIX Security Symposium*, 2023.

[47] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "Maze: Towards automated heap feng shui." in *USENIX Security Symposium*, 2021.

[48] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 781–797. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei

[49] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 71–88.

[50] B. Zhang, J. Chen, R. Li, C. Feng, R. Li, and C. Tang, "Automated exploitable heap layout generation for heap overflows through manipulation distance-guided fuzzing," in *USENIX Security Symposium*, 2023.

[51] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 811–824.

[52] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System Call-Based State-Aware linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3273–3289. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-bodong

[53] X. Zou, Y. Hao, Z. Zhang, J. Pu, W. Chen, and Z. Qian, "SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem," in *31st Annual Network and Distributed System Security Symposium, NDSS*, 2024.

[54] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "{SyzScope}: Revealing {High-Risk} security impacts of {Fuzzer-Exposed} bugs in linux kernel," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3201–3217.