KALD: Detecting Direct Pointer Disclosure Vulnerabilities

Brian Belleville, Wenbo Shen, Stijn Volckaert, Ahmed M. Azab, and Michael Franz, *Fellow, IEEE*

Abstract—Modern operating system kernels deploy Kernel Address Space Layout Randomization (KASLR) to mitigate control-flow hijacking attacks. KASLR randomizes the base addresses of the kernel's code and data segments. However, it randomizes both with a single offset and does not randomize the internal layout of either of these segments, so relative addresses remain known to adversaries. If the kernel discloses a single code or global data pointer, an adversary can therefore infer the entire layout of the kernel's code segment and bypass KASLR. In this paper, we present Kernel Address Leak Detector (KALD), a tool that finds direct disclosure vulnerabilities by statically analyzing the kernel source code. KALD can analyze the source code of modern operating system kernels and find previously unreported leaks. KALD compiles a list of functions that can leak information to user-space accessible locations, and it uses the results of a points-to analysis to determine whether individual invocations of such functions can disclose kernel pointers. We show that KALD successfully detects several direct disclosure vulnerabilities in the Linux kernel and that it is flexible enough to be useful in practice.

Index Terms—Program analysis, static analysis, address leakage, security, operating systems security, LLVM, Linux kernel

1 Introduction

This year marks the 30 year anniversary of the Morris Worm, the first significant exploit of a buffer overflow vulnerability [1]. Although much progress has been made on techniques that mitigate memory corruption, many techniques have failed to see widespread deployment due to performance concerns [2]. Randomization-based defenses are among the techniques that *have* found or that are finding their way into commodity systems, thanks in no small part to their relatively low run-time performance impact.

Since memory corruption exploits typically require knowledge of the memory layout, randomization-based defenses force adversaries to expend additional effort to learn the memory layout before attempting to take control of the victim process. In the simplest case, the adversary seeks to overwrite a specific variable in the program. With randomization, the attacker must first guess or discover the address in memory used to store that variable. If the adversary resorts to guessing, odds are that the program will crash and thus alert defenders before a correct guess is made. Since randomization relies on hiding the memory layout, adversaries look for ways to leak the memory contents as an alternative to raw guessing [3], [4].

Randomization can be applied at many granularities and at different stages in a program's development and deployment cycle [5]. Address space layout randomization (ASLR) is the de facto standard way to add randomness to the memory layout of a running process [6]. ASLR shifts the base address of each segment (code, data, stacks) by adding a random offset to its default location. Since ASLR preserves the internal structure of each segment, an adversary that leaks a single pointer to a section can infer the exact layout of that entire section.

ASLR was first adopted for code running in user mode, but is now commonly applied to operating system kernels as well. The Linux kernel, which is used for desktop PCs, servers, and mobile phones (as part of the Android operating system), now supports kernel ASLR (KASLR). KASLR functions similarly to ordinary ASLR for user-space programs. During system boot, the locations of code and data are placed at a randomized offset within the available kernel memory space.

KASLR independently randomizes the base addresses of the kernel, stack, and heap at boot time, but it does not randomize the internal layout of any of the regions. Furthermore, KASLR uses the same randomization offset for the code and global data within the kernel region. Attackers can therefore infer the entire layout of the kernel, including that of its executable code, if they can discover a single pointer to a known instruction or global data variable. Since pointers to kernel code or global data must be kept secret, we refer to such pointers as *sensitive* pointers.

If attackers discover a sensitive pointer, and thus learn the kernel code layout, they could then attempt to mount privilege escalation attacks against the kernel using either code reuse or non-control data corruption. A Google Project Zero member recently demonstrated a return-oriented programming attack against Android smartphone kernels by

- *B. Belleville and A. M. Azab are with Google. E-mail: {bbelleville, amazab}@google.com.*
- *W. Shen is with the Institute of Cyberspace Research, Zhejiang University, 38 Zheda Road, Xihu District, Hangzhou, Zhejiang Province 310027, China. E-mail: shenwenbo@zju.edu.cn.*
- *S. Volckaert is with imec-DistriNet, Department of Computer Science, KU Leuven, Leuven 3000, Belgium. E-mail: stijn.volckaert@cs.kuleuven.be.*
- *M. Franz is with the University of California, Irvine, CA 92697, USA. E-mail: franz@uci.edu.*

Manuscript received 29 Nov. 2018; revised 15 Mar. 2019; accepted 1 May 2019. Date of publication 10 May 2019; date of current version 13 May 2021. (Corresponding author: Brian Belleville).
Digital Object Identifier no. 10.1109/TDSC.2019.2915829

TABLE 1
ASLR Adoption in Mainstream Operating Systems

Operating System	ASLR Type	Year
Linux 2.6.12	User	2005
Windows Vista	User/Kernel	2007
OS X 10.5	User	2007
iOS 5	User	2011
Android 4.0	User	2011
OS X 10.8	Kernel	2012
iOS 6	Kernel	2012
Linux 3.14	Kernel	2014
Samsung Android 6.0	Kernel	2016
Android 8.0	Kernel	2017

exploiting overflow bugs in a touch screen driver [7]. At BlackHat 2017, a security researcher presented a privilege escalation attack on Android by combining a jump-oriented programming attack to get a root process and a data-only attack to disable SE-Android [8].

The first step in both of these attacks is leaking a kernel pointer. A number of recent vulnerabilities show that it is feasible to leak such pointers to non-privileged applications running in user-space. These vulnerabilities exist in two forms: those that exploit memory corruption to disclose pointers [8], [9], [10], [11], [12], [13], and those where the kernel code directly discloses addresses [14], [15]. There are also attacks that infer kernel-space addresses indirectly through micro-architectural side channels [16], [17], [18]. While micro-architectural side channels can be prevented using kernel page table isolation [19], and there are many strategies to statically detect memory corruption vulnerabilities within the kernel [20], [21], [22], there are few mitigations that prevent or detect direct disclosure vulnerabilities.

In this paper, we present the Kernel Address Leak Detector (`KALD`), a static analysis tool that can find locations where the kernel directly leaks sensitive addresses to user-space memory. Contrary to the current practice of labor-intensive and error-prone manual code reviewing, our tool can automatically analyze the kernel source code to find these issues.

2 Background

2.1 Code-Reuse Attacks

Commodity operating systems primarily rely on enforcement-based techniques to prevent certain types of exploits. Enforcing strict data execution prevention (W$\oplus$X) prevents most code-injection attacks [23], as pages cannot be simultaneously writable and executable. Modern operating systems also prevent executing user-space code with kernel privileges by using supervisor mode execution prevention on x86 [24] and privilege execute-never on ARM [25]. Unfortunately, attackers can circumvent these mitigations by crafting exploits that reuse existing kernel code [26], [27]. Code-reuse attack techniques exploit memory corruption vulnerabilities to alter control flow data such that existing sequences of instructions (a.k.a. *gadgets*) are chained together in order to perform malicious actions. These techniques include return-oriented programming (ROP) [28], which targets return addresses on the stack, and jump-oriented programming (JOP) [29], which targets function pointers.

2.2 Kernel ASLR

One common defense against code-reuse attacks is address space layout randomization. ASLR introduces randomness into the memory layout of the program. For example, user-space ASLR on Linux randomizes the location of stack, mmap, brk and text sections. Similar to user-space ASLR, KASLR is implemented in modern operating systems by adding a random offset, the *KASLR offset*, to the default loading address of kernel, so that the location of the kernel's code and global data gets randomized during system startup. This stops kernel code-reuse attacks because an attacker no longer knows the exact address of any gadgets. An attacker will now have to guess the location of the gadgets, and if the layout entropy is large enough, it is very likely that the guessed locations will be wrong, and the attack will fail.

User and kernel ASLR are effective countermeasures to code-reuse attacks and are widely adopted by mainstream operating systems [30], as shown in Table 1.

2.3 Address Leakage

The most common technique to bypass ASLR is code derandomization using address leakage. When KASLR is applied, the first step towards a successful code-reuse attack is to bypass KASLR by leaking a code or global data pointer [8].

There are several ways that an attacker can find such a pointer. One option is to exploit a memory disclosure vulnerability to read a kernel address directly from the kernel memory. However, this requires finding a reliable memory disclosure, which may not be available. Another option is to find a case where the kernel writes an address to a user-readable location. This could be user-space memory, if the kernel writes the value of a kernel pointer there as the result of a system call, or it could be a user-readable log file, if the kernel outputs addresses as part of log messages as in recently discovered vulnerabilities [14], [15]. Most of the kernel's code was written before KASLR was implemented, and there are cases where the kernel discloses addresses to user processes. As a result, it is often possible to find a kernel address. Kernel developers now know that kernel addresses should not be written to user-space memory or logged, but without an effective way to catch these issues, there may still exist cases where addresses are written.

3 Design

We designed `KALD` as a tool that statically analyzes the kernel source code to find code that may leak the KASLR offset. Since `KALD` performs static analysis, it can achieve complete program coverage and detect leaks even on rarely executed code paths. This is not possible with alternative methods like fuzzing or taint analysis, which can only find issues on code that is actually executed. Fig. 1 illustrates the different steps of this static analysis. `KALD` begins by finding and marking all calls to output functions. These are functions that can write data to user-visible locations, such as the register context or user-space memory.

Next, `KALD` runs a points-to analysis to calculate the set of memory locations each program value may point to. `KALD`

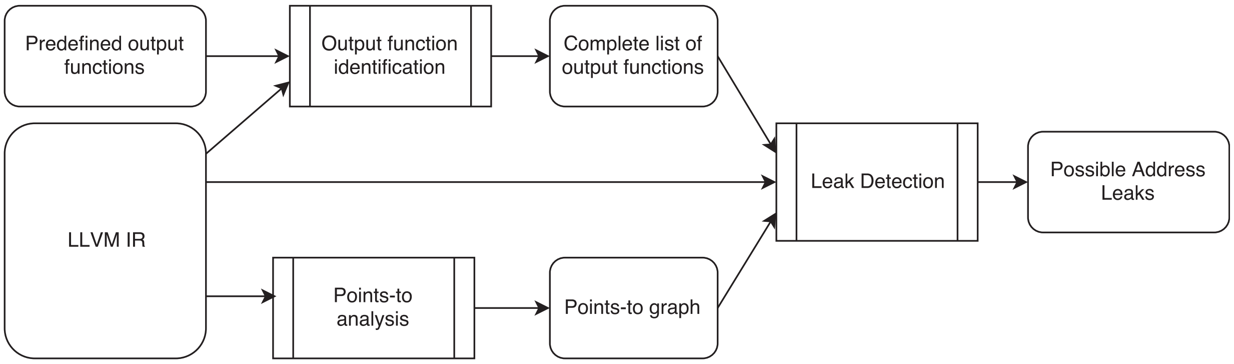

Predefined output functions
Output function identification
Complete list of output functions
LLVM IR
Leak Detection
Possible Address Leaks
Points-to analysis
Points-to graph

Fig. 1. Sequence of operations in `KALD`.

then examines all calls to output functions and queries the results of the points-to analysis to determine if any of the function arguments passed at the call site are sensitive values. If a call to an output function may leak sensitive values, then we mark the call site as potentially dangerous.

3.1 Output Function Identification

While the kernel can directly write data to any location, it is customary and highly advisable to use the standardized interface when accessing user-space memory. This interface ensures that the currently running process can legally access a location, and that no memory errors occur (e.g., because the target page is swapped out to disk). The interface includes API functions such as `copy_to_user`, which can copy data to any virtual memory page mapped into the user-space program, as well as functions to write to the `proc` file system, functions to write to the user-space register context, etc.

We compiled a list of these standardized functions, and use this list to seed our output function analysis. The list we used in our evaluation is shown in Table 2. For each function in the list, we keep track of its name, location, as well as all the necessary data needed by later stages of our analysis such as which arguments the function copies to user memory, and whether the arguments are copied as a value or dereferenced as a pointer. In this list, we also indicate if the function may perform string formatting, and, if so, which fixed argument contains the format string. Adding support for an additional output function only requires adding the function name and argument information to this list.

We then analyze the kernel source code to find additional calls to output functions. We are specifically interested in calls to format string functions such as `printk`. Some of the format string functions write data into the system log file, which can be accessed from user-space. These functions can leak the KASLR offset, but they only leak pointers printed out using certain format specifiers (e.g., the integer format specifier `%ld`, or the pointer format specifier `%p`).

Many of the other format specifiers are harmless, as they cannot reveal the literal value of a pointer argument. The `%s` specifier, for example, indicates that the pointer argument points to a string, and that the string will be printed but the pointer itself will not be. Linux's core format string functions also support the `%pK` specifier, which does print out pointers, but obfuscates their value if the pointer points to kernel-space memory, and the currently running user-space program does not have sufficient privileges.[1]

Our output function identification step attempts to parse format strings to identify pointer arguments that could leak to user-accessible locations. This is usually possible because most format strings are constants, and can thus be parsed at compile time. For non-constant format strings, we conservatively assume that all of the pointer arguments to the format string function can potentially leak to user-space.

TABLE 2
Output Functions

`printk`
`printk_once`
`printk_ratelimited`
`seq_printf`
`copy_to_user`
`_copy_to_user`
`__arch_copy_to_user`
`put_user`
`__put_user_1`
`__put_user_2`
`__put_user_3`
`__put_user_4`
`__put_user_8`

Listing 1. A minimal example using the the `%pV` format specifier

```
void my_printk(struct device* dev,
               const char* fmt, ...)
{
 struct va_format vaf;
 va_list args;
 va_start(args, fmt);
 vaf.fmt = fmt;
 vaf.va = &args;

 if (dev) {
   printk("%s: %pV", dev_name(dev), &vaf);
 } else {
   printk("(no dev):%pV", &vaf);
 }

 va_end(args);
}
```

1. Specifically, in order to see the real pointer values, the process must have the `CAP_SYSLOG` capability, and it must not have changed its `uid` since it originally started.

One particularly challenging aspect of distinguishing harmless format string function calls from potentially dangerous calls is the Linux kernel-specific %pV format specifier, which is used to perform recursive string formatting. The corresponding argument of this format specifier is a pointer to a va_format structure. This struct contains two fields: a pointer to a format string, and an argument list. The format string function recursively substitutes the %pV specifier by the format string in its corresponding va_format structure, and it also substitutes the corresponding argument itself by the argument list specified in the va_format structure. Listing 1 shows a minimal code example that uses the %pV format specifier.

Our analysis inspects every call to a format string function, parses the format string, and searches for the %pV format specifier. When that format specifier is found, KALD identifies the caller of the function, and determines the source of the format string and argument list. We then add the call to the format string function to our list, along with the expanded format string and the expanded function argument lists.

The output function identification step of KALD is Linux-specific. KALD could be ported to analyze other operating system kernels by identifying the relevant API functions within those operating systems, and handling any platform-specific behavior of these functions.

3.2 Points-to Analysis

KALD performs a full-program points-to analysis. Points-to analysis is a static program analysis that computes an approximation of the set of objects that a pointer can refer to [31]. We use a field-sensitive version of Andersen's algorithm [32] because there is a high-quality implementation available for LLVM [33]. Our tool can theoretically work with any points-to analysis, as long as the analysis can provide the set of objects referred to by each pointer. KALD could, in other words, easily be adapted to use a different algorithm, allowing it to be improved if a more sophisticated algorithm is available.

3.3 Leak Detection

After the points-to analysis completes, KALD inspects each function call site to determine if it could leak any sensitive addresses. If the call site targets a function in the list of output functions, KALD checks each of the arguments to determine if any may leak a sensitive address. If an argument may leak a sensitive address, the call site is flagged as a potential leak location. The list of output functions indicates whether the output argument is used directly as a value, as in the case of the first argument of put_user, or dereferenced as a pointer, for example the second argument of copy_to_user. KALD uses this information to properly analyze the call site.

If the argument is used as a value, KALD will indicate a potential leak if it is a pointer containing a sensitive address. If the argument is dereferenced, KALD will examine the pointed-to objects and will indicate a potential leak if any of these objects may be a sensitive pointer, or if any are a struct or buffer which may contain a sensitive pointer.

As we described in Section 3.1, calls to format string functions require special treatment as the format string itself determines whether the function may output pointer values. KALD parses this format string during the output function identification step (cf. Section 3.1), and identifies which arguments may leak sensitive address values.

Using the results of the points-to analysis and the list of functions that can copy data to the user, KALD can check if calls to output functions may leak addresses to user programs. However, analyzing a large code base such as the Linux kernel using a sound may-alias analysis can result in false positives if an overly-conservative points-to result says that a pointer may reference a large number of objects. In order to limit false positive rate, we apply a type-based heuristic to the points-to results. We use the actual pointer type available in the code, and compare it with the set of objects that the points-to results indicate may be referenced by the pointer. If the types do not match, then we remove that element from the results. To avoid overly restricting the points-to set, we do not require that the types are exact matches. For scalar types we allow any type that can be losslessly cast to the target type. For struct types we use the common initial sequence criteria. As defined by Yong et al., two structures have a common initial sequence if, for a sequence of initial fields, all corresponding fields have compatible types [34]. If the type of the actual argument is a common initial sequence of the type of the referenced object, then we consider the types to match. For array types we apply the relevant criteria to the array element type, including if the element type is itself an array type, thereby handling nested array types.

Since we use a field-sensitive analysis, we are also able to apply this heuristic to pointers that may refer to sub-fields of global structures. The points-to analysis provides the index into the referenced object. If the type of the field at that index is compatible with the type of the actual argument, then we consider the types to be compatible.

The type-based heuristic allows us to successfully limit the results of the points-to analysis, but there are cases where the type information is either not available, or is not useful. This is the result of generic pointers (void*) and character pointers (char*), which may point to objects of any type, even under the strict aliasing rule in C. For generic pointers, we conservatively assume that any type is a possible target, and do not limit the results based on type.

Using a type-based heuristic can result in false negatives, particularly for a program that is not type-safe. As a result, KALD will not detect an address leak that is the result of type confusion. We do not assume that the Linux kernel is completely type safe, but we consider this a worthwhile trade-off to reduce the number of false positives. Other kernel static analysis tools have also found this trade-off to be desirable [20]. Finding type confusion bugs is outside the scope of this work.

4 Implementation

We implemented KALD as an analysis pass that operates on LLVM Intermediate Representation (IRr) code [35]. We use Clang, the C/C++ frontend of the LLVM project, to compile C source code into LLVM IR, and llvm-link, the LLVM linker, to link multiple LLVM IR files into a single IR file.

KALD utilizes pointer analysis to detect pointers to kernel code and data. We used the implementation of Andersen's analysis [32] from the open source SVF framework [33] as the underlying pointer analysis of our implementation.

TABLE 3
Summary of Results

Total reported issues	408
True issues	73
Total analysis CPU time	23h 43m
Maximum analysis wall clock time	15h 19m
Median analysis wall clock time	1.8 s
Average analysis wall clock time	9m 57s
Maximum memory use	97.8 GB
Median memory use	305 MB
Average memory use	3.0 GB

The input to KALD is the LLVM IR of a program. KALD operates as a module pass, analyzing the entire LLVM module it is given as input. We analyze each call site that appears in the LLVM IR to determine if it could potentially leak a kernel address based on the function being called and the arguments at that call site. However, there are also some kernel interfaces for writing data to user-space that may be implemented with inline assembly code. For example, on both x86 and ARM64, put_user is implemented as a C macro that inserts inline assembly code. KALD handles these cases by recognizing the assembly code for these operations. Within LLVM IR, inline assembly appears as a call site targeting an InlineAsm value. The InlineAsm object contains the string of the assembly code that will be inserted. KALD will check for inline assembly that could leak information by matching the assembly code with a set of known assembly code strings.

KALD can process IR files of any size, but performing global analysis on large code bases can be time and memory intensive. To improve the usability of KALD for frequent analysis during the development cycle, KALD has the option to limit the analysis time. However, limiting the analysis time can result in false negatives (missed issues) if the analysis does not converge within the allowed time. Alternatively, users can reduce the analysis time by limiting the size of the input by running KALD incrementally on submodules of a program individually. Smaller programs will be more likely to converge within the allowed time, however, this may also introduce false negatives since the complete program was never analyzed as a whole. KALD does not try to model any code that is not available for analysis, so when analyzing submodules individually, KALD can find issues resulting from behavior within the code being analyzed, but will not detect leaks that only result from cross-module interactions.

Limiting the analysis time or analyzing submodules of a program is a source of false negatives, but not of false positives. A false negative may occur when a pointer could contain a sensitive address, but the analysis is unable to process the relevant assignments to determine the full points-to set, either due to exceeding the time budget or not having all the necessary code made available for analysis. However, this cannot cause false positives because limiting the analysis can only result in smaller points-to sets that may not contain all of the possible targets for a pointer.

5 Results

We evaluated KALD by running it on the source code of Linux 4.14, with patches applied to make it fully compatible with Clang [36]. To maximize the coverage of our analysis, we built the kernel with the maximal set of configuration options enabled (allyesconfig) for x86_64. We used Whole Program LLVM[2] to automatically collect the LLVM IR files created by Clang and link them together. In our evaluation we analyzed the kernel code on a per-subdirectory basis. For each top-level kernel subdirectory other than drivers, we analyzed all of the code within that directory in one pass. For the drivers subdirectory, due to the large amount of code, we analyzed each subdirectory within drivers individually. We analyzed a total of 143 modules and KALD identified 408 potential address leaks. These results are summarized in Table 3. The issues reported by KALD were manually verified by source code analysis. We analyzed all potential address leaks found by our tool to determine the number of true issues, and found that in 73 of the cases a kernel address could leak. The remaining cases were false positives. We measured the running time and peak memory use of the analysis of each of the 143 modules, and report the maximum, median, and average of the 143 uses of KALD.

TABLE 4
Summary of Issues Discovered

`printk`	66
custom format functions	6
`copy_to_user`	1

The issues that KALD discovered are summarized in Table 4. Most address leaks identified were uses of printk. In addition to printk, there were also 6 issues from uses of custom string formatting functions. These functions were identified by the output function identification step described in Section 3.1. There was also one issue resulting from a use of copy_to_user. We discuss this vulnerability in depth in Section 5.1.

The sources of false positives are presented in Table 5. The primary sources of false positives are cases where the analysis is overly conservative and determines that the pointer can refer to a large number of objects, and there is no type information available to reduce the number of targets. This corresponds to the large over-approximation row in Table 5. In these cases it is difficult to determine the actual referents of the pointer.

Another common source of false positives is the use of intrusive linked lists throughout the kernel. This is the struct list_head embedded in many kernel data structures. These implement doubly-linked, circular, linked-lists. In many cases the list is accessed through global variable, but the global element is used only as a list handle. In these cases the analysis conservatively reports that a pointer to an element in the list may refer to the global value, and hence leak the KASLR offset.

More minor sources of false positives include our conservative handling of non-constant format strings and lack of bounds information for some structure copies. As described in Section 3.1, if the format string is not known at compile-time we conservatively assume that all arguments may leak. Generally this is an overly conservative assumption. The analysis also does not attempt to determine the bounds

2. https://github.com/travitch/whole-program-llvm

of memory copies. This means that a partial copy of a struct that contains a sensitive pointer will lead to a false positive.

We reported exploitable leaks to Linux kernel developers. Independently of this work, kernel developers added a mitigation for these issues within their string formatting implementation. Starting with Linux 4.15, the `%p` format specifier will hash the pointer value and print a unique identifier that can not leak the KASLR offset [37]. This shows that these address leaks are serious enough to warrant mitigation.

5.1 Kernel Address Leaks Detected by `KALD`

Listing 2. Simplified source code resulting in a leak of the KASLR offset through `printk`

```
static struct uart_port cdns_uart_port[2];

int
uart_remove_one_port(struct uart_driver *drv,
                     struct uart_port *uport) {
 struct uart_state *state;
 struct uart_port *uart_port;
 state = drv->state + uport->line;
 uart_port = uart_port_check(state);
 if (uart_port != uport)
  dev_alert(uport->dev,
      "Removing wrong port: %p != %p\n",
      state->uart_port,
      uport);
}

void
dev_alert(const struct device *dev,
    const char *fmt, ...) {
 struct va_format vaf;
 va_list args;
 va_start(args, fmt);
 vaf.fmt = fmt;
 vaf.va = &args;
 if (dev) {
  /* elided */
 } else
  printk("%s(NULL device*): %pV", level, vaf);
 va_end(args);
}

static int
cdns_uart_remove(struct platform_device *pdev) {
 struct uart_port *port = pdev->dev.driver_data;
rc=uart_remove_one_port(&cdns_uart_uart_driver,
                        port);
 return rc;
}

static int
cdns_uart_probe(struct platform_device *pdev) {
 struct uart_port *port;
 int id;
 id=of_alias_get_id(pdev->dev.of_node, "serial");
 port = &cdns_uart_port[id];
 pdev->dev.driver_data = port;
 return 0;
}
```

In this section we examine two real address leaks that were found using `KALD`. These are previously unreported leaks, and are presented to highlight how different aspects of our design help to discover leaks.

The first is a leak of an address through a call to `printk`. Listing 2 shows a simplified version of the vulnerable code. Within function `uart_remove_one_port`, `KALD` indicates that the fourth argument to `dev_alert` may leak the global variable `cdns_uart_port`. The function `dev_alert` is an output function that is not in our initial list of known output functions, but was detected automatically in the output function identification step. Within `dev_alert`, `printk` is called using the `%pV` format specifier. The `uport` pointer comes from the caller of `uart_remove_one_port`. This function is called by `cdns_uart_remove`, which gets the port from a field within a structure. This field is set within `cdns_uart_probe`, and is set to be a pointer to an element within `cdns_uart_port`. Therefore, when `uart_remove_one_port` is called from `cdns_uart_remove`, the address of a global variable can be leaked.

This example shows the combined benefits of our output function identification step and the interprocedural points-to analysis. The output function that leads to the leak is not in the predefined list of output functions, but `KALD` automatically adds it in the first step of its analysis. The interaction that leads to the leak is the result of interprocedural behavior of multiple functions. Detecting this leak requires interprocedural analysis.

Listing 3. Simplified source code of a leak of the KASLR offset through `copy_to_user`

```
static struct floppy_struct floppy_type[32] = {
 {  0, 0,0, 0,0,0x00,0x00,0x00,0x00,NULL },
 { 720, 9,2,40,0,0x2A,0x02,0xDF,0x50,"d360" },
 /* elided */
};

static int
fd_ioctl(struct block_device *bdev, fmode_t mode,
    unsigned int cmd, unsigned long param) {
 int type = ITYPE(UDRS->fd_device);
 int size = sizeof(struct floppy_struct);
 const void *outparam;

 switch (cmd) {
 case FDGETPRM:
  outparam = &floppy_type[type];
  break;

 default:
  return -EINVAL;
 }

 copy_to_user((void *)param, outparam, size);
 return 0;
}
```

The other leak occurs due to a call to `copy_to_user`, as shown in Listing 3. This vulnerability was confirmed by kernel developers and assigned CVE-2018-7755 [38]. It was patched in Linux 4.19 [39]. Function `fd_ioctl` implements

TABLE 5
Source of False Positives

Large over-approximation	214
Linked lists	38
Non-constant format string	17
Bounds	11
Other	55

the ioctl interface for floppy disk drives. This driver contains the FDGETPRM ioctl, which copies a floppy_struct to user-space address param. The floppy_struct exists in the global array floppy_type, so the address of the struct itself is a sensitive address. However, when used as an argument to copy_to_user, the pointer is not output directly. Instead, the memory it refers to is copied. When KALD analyzes a call to copy_to_user, it checks if the object referenced by the second argument contains any pointers. In this case it does contain a sensitive pointer. The final field of a floppy_struct is a pointer, a char*, which points to a string representing the name of the device. The elements of the array floppy_type are initialized with pointers to constant strings within the kernel's memory. When the copy_to_user executes, this kernel address will be leaked to user-space memory.

KALD can detect this leak since it models the behavior of different output functions and distinguishes between pointers output directly and pointers that are dereferenced. This, in concert with the points-to analysis, allows KALD to determine that in this case the copy_to_user call will leak a kernel address.

We empirically verified this leak by sending the FDGETPRM ioctl to the floppy driver on a virtualized system with a virtual floppy drive. The returned struct does contain a kernel pointer. Across multiple boot cycles on a system with KASLR enabled, the relative offset between the returned pointer and kernel code pointers remained unchanged. This demonstrated that the returned pointer was randomized with the KASLR offset and could be used to bypass KASLR.

5.2 Finding Known Leaks

We also verified that KALD can detect previously reported address leaks. We identified two recently reported direct pointer disclosures, obtained the kernel versions containing the vulnerabilities, and analyzed the vulnerable code with KALD.

The first was a leak of static variables through a call to seq_printf within the pm_qos module [14]. The function seq_printf is a string formatting function that can be used to format output to proc files. This leak is the result of a programmer mistakenly using the format string %pk with a lower case k for a kernel pointer instead of the correct %pK. While this could be confusing for a human reviewer, KALD correctly parses the format string and determines from the points-to analysis that the argument is a kernel pointer. Since the analysis is able to determine that this call could leak a kernel pointer, KALD reports a potential leak.

The other leak we examined was a printk call within the function acpi_smbus_hc_add. In this case, the leaked address was the address of a struct acpi_ec. These structs are allocated from the kernel heap in the function acpi_ec_alloc. Since this call site can only leak a heap address, KALD does not report a potential issue. Heap addresses are outside the scope of what KALD is designed to detect because the heap is not randomized with the same KASLR offset as the code. Finding a heap address will not provide an attacker with the information they would need to construct a code reuse exploit. However, the pointer analysis used by KALD tracks heap objects, so not detecting heap address leaks is not a limitation of our approach, but rather an intentional design choice in order to focus on address leaks that are most likely to be useful in an attack.

6 Limitations

KALD requires whole program interprocedural points-to analysis, which is both time and memory intensive, especially for programs as large as the Linux kernel. There have been recent proposals to allow interprocedural program analyses to scale to programs the size of the Linux kernel. These include using a disk-based graph solving system to overcome memory limitations [40] or partitioning the kernel along system call boundaries in order to decompose the analysis to elements of a manageable size [21]. KALD could utilize these techniques to lower the time and memory requirements of the pointer analysis phase of operation.

A conservative points-to analysis can also lead to a high false positive rate, especially for code bases as large and complex as Linux. A more precise pointer analysis will reduce the false positive rate, but that could further increase the time and memory required for analysis.

The issue of false positives can also be mitigated by improving the usability of the tool and making it easier for users to identify true issues. This can be done by creating heuristics to sort, filter, and prioritize results to show the issues most likely to be true. The tool could also be extended in order to make it easier for developers to verify reported issues by providing a more detailed report indicating which program statements resulted in the points-to result that is reported.

7 Related Work

7.1 Static Analysis

KINT uses static analysis to find integer bugs in C programs [41]. KINT successfully identified several exploitable bugs in the Linux kernel. Similar to KALD, KINT uses heuristics to prioritize and filter its analysis results.

APISAN builds a database of (likely) correct API usage patterns by symbolically executing code and inferring semantic relationships between API calls [42]. APISAN can then statically analyze additional code and cross-check its API usage patterns with the database.

DR. CHECKER is a static bug finding tool for the Linux kernel [20]. It perform points-to and data-flow analysis, and has several clients that can find different sorts of violations. None the DR. CHECKER's built-in clients find direct pointer disclosure vulnerabilities, however. The clients also only support analysis of kernel drivers, while KALD can analyze the complete kernel source code. DR. CHECKER sacrifices soundness by limiting the number of iterations of the analysis. This allows the tool to maintain a reasonable run time and reduces the false positive rate. KALD also attempts to reduce the false positive rate, although our approach is to use heuristics to filter certain results.

K-Miner statically analyzes the Linux kernel source code to identify memory corruption vulnerabilities [21]. Similar to KALD, K-Miner builds on top of LLVM and the SVF analysis framework. K-Miner catches issues that other analysis frameworks cannot detect because they lack interprocedural data flow graphs.

7.2 Direct Disclosure Detection

UniSan seeks to preserve the integrity of the KASLR offset [22]. It focuses on preventing leaks through uninitialized values, while our work finds errors where developers have chosen to output kernel addresses. UniSan has both an analysis component that identifies possible issues, and a run-time protection component that mitigates them. Our solution is solely an analysis tool that reveals the issues to the developers.

7.3 Improved KASLR

kR̂X applies fine-grained address space layout randomization to the Linux kernel and enforces execute-only memory on kernel code [43]. Return addresses are protected from leaks using decoy addresses or XOR encryption, but function pointers are not protected. However, the fine-grained randomization randomizes function layout and ordering, so leaking a single function pointer does not disclose the entire kernel's layout.

OpenBSD supports kernel address space randomized link (KARL) [44]. KARL randomizes the link order of all object files used to build the OpenBSD kernel. Every time the kernel is booted, a new version is linked and installed. This provides a greater degree of randomization than KASLR, as a leaked address now only provides information about the offset of a single object file, but not about the complete kernel. Additionally, since a new randomized version is linked on each boot, an attacker cannot perform offline analysis of the kernel's layout.

8 Conclusions

Pointer disclosure vulnerabilities can be exploited by adversaries to bypass KASLR. While there have been efforts to address sources of pointer disclosures, these techniques do not address detect direct disclosure vulnerabilities where the kernel outputs code pointers on purpose.

We presented the Kernel Address Leak Detector (KALD), a tool that statically analyzes the Linux kernel source code to detect direct pointer disclosure vulnerabilities. KALD compiles a list of functions that can leak information to user-space programs, and uses the results of a points-to analysis to determine whether specific invocations of these functions can disclose kernel code pointers based on the arguments passed to the function. KALD detected several direct disclosure vulnerabilities in the latest version of the Linux kernel.

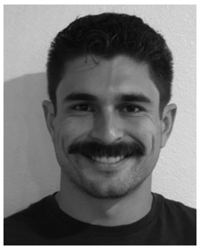
Acknowledgment

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, and by the National Science Foundation under awards CNS-1619211 and CNS-1513837. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government. When this research was conducted, B. Belleville was with the University of California, Irvine, and A.M. Azab and W. Shen were with Samsung Research America. This research started while B. Belleville was an intern at Samsung Research America, mentored by W. Shen and A.M. Azab.

References

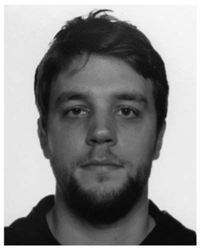
[1] E. H. Spafford, "The internet worm program: An analysis," *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, pp. 17–57, Jan. 1989. [Online]. Available: http://doi.acm.org/10.1145/66093.66095
[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 48–62.
[3] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proc. 2nd Eur. Workshop Syst. Secur.*, 2009, pp. 1–8.
[4] F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012. https://ieeexplore.ieee.org/abstract/document/6547134
[5] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 276–291.
[6] PaX Team, "PaX ASLR," 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt
[7] Google Project Zero, "Multiple overflows in tsp sysfs "cmd store"," 2016. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=967
[8] D. Shen, "Defeating samsung knox with zero privilege," *BlackHat USA*, 2017.
[9] "CVE-2015–8569," 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015–8569
[10] "CVE-2015–1097," 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015–1097
[11] "CVE-2017–14954," 2017. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017–14954
[12] "CVE-2017–1000410," 2017. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017–1000410
[13] "CVE-2015–0089," 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015–0089
[14] "Samsung: Kaslr bypass in "pm_qos"," 2016. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=971
[15] "CVE-2018–5750," 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018–5750
[16] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 380–392.
[17] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 191–205.
[18] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 368–379.
[19] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Proc. Int. Symp. Eng. Secure Softw. Syst.*, 2017, pp. 161–176.
[20] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," in *Proc. 26th USENIX Conf. Secur. Symp.*, 2017, pp. 1007–1024.
[21] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 2018.

[22] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 920–932.
[23] Microsoft, "Data execution prevention (DEP)," 2006. [Online]. Available: http://support.microsoft.com/kb/875352/EN-US/
[24] Intel, Inc. Intel® 64 and IA-32 Architectures Software Developer's Manual. No. 325462-069US, Jan. 2019.
[25] ARM Limited. ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. ARM DDI 0487D.b, 2019.
[26] W. Xu and Y. Fu, "Own your android! yet another universal root," in *Proc. USENIX Workshop Offensive Technol.*, 2015.
[27] "evasi0n iOS 7.0.x jailbreak - official website of the evad3rs," 2013. [Online]. Available: http://evasi0n.com/
[28] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2007, pp. 552–561.
[29] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2010, pp. 559–572.
[30] Y. Jang, S. Lee, and T. Kim, "Drk: Breaking kernel address space layout randomization with intel tsx," *Proc. 2016 ACM SIGSAC Conf. Comp. Commun. Secur.*, 2016, no. 13, pp. 380–392, http://doi.acm.org/10.1145/2976749.2978321.
[31] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, pp. 1–69, 2015. [Online]. Available: http://dx.doi.org/10.1561/2500000014
[32] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, DIKU, Univ. Cophenhagen, København, Denmark, 1994.
[33] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proc. 25th Int. Conf. Compiler Construction*, 2016, pp. 265–266.
[34] S. H. Yong, S. Horwitz, and T. Reps, "Pointer analysis for programs with structures and casting," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1999, pp. 91–103.
[35] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, 2004, Art. no. 75.
[36] M. Kaehlcke, "Clang patch stacks for lts kernels and status update," Nov. 2017. [Online]. Available: https://lkml.org/lkml/2017/11/22/943
[37] T. C. Harding, "hash addresses printed with %p," Nov. 2017. [Online]. Available: https://lwn.net/Articles/740249/
[38] "CVE-2018–7755," 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018–7755
[39] J. Axboe, "floppy: Do not copy a kernel pointer to user memory in fdgetprm ioctl," Sep. 2018. [Online]. Available: https://lkml.org/lkml/2018/9/20/742
[40] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 389–404.
[41] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with kint," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 163–177.
[42] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing api usages through semantic cross-checking," in *Proc. USENIX Secur. Symp.*, 2016.
[43] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kR̂X: Comprehensive kernel protection against just-in-time code reuse," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 420–436. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064216
[44] J. Edge, "OpenBSD kernel address randomized link," Jul. 2017. [Online]. Available: https://lwn.net/Articles/727697/



Brian Belleville received the PhD degree from the University of California, Irvine, in 2018. He is a software engineer with Google. His interests include systems security, compilers, and programming languages.

Wenbo Shen received the bachelor's degree from Harbin Institute of Technology, Harbin, China, in 2010 and the PhD degree in computer science from North Carolina State University, Raleigh, in 2015. He is currently a ZJU 100 young professor with Zhejiang University. His research areas include operating system security, container security, and program analysis. Before joining Zhejiang University, he was a member of the Samsung Knox Kernel team at Samsung Research America, Mountain View, California.



Stijn Volckaert received the PhD degree in computer science engineering from Ghent University. He spent more than two years as a postdoctoral scholar with the Department of Computer Science, University of California, Irvine, and is now an assistant professor with the Department of Computer Science, KU Leuven. His research interests include systems security, fault tolerance, software diversity, and software protection.

Ahmed M. Azab received the PhD degree from North Caroline State University, in 2011. He is a senior software engineer with Google. Previously, he worked with Samsung Research America. For over a decade he has developed techniques that prevent or mitigate dangerous forms of cyber security attacks, such as malicious code injection and malicious code reuse.

Michael Franz received a Dr Sc Tech degree in computer science and the Dipl Informatik-Ing ETH degree, both from the Swiss Federal Institute of Technology, ETH Zurich. He is a chancellor's professor with the Department of Computer Science, Donald Bren School of Information and Computer Sciences, University of California, Irvine. He is a fellow of the Association for Computing Machinery (ACM) and a fellow of the Institute of Electrical and Electronics Engineers (IEEE).