Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels

Wenwen Wang, Kangjie Lu, and Pen-Chung Yew
University of Minnesota, Twin Cities

ABSTRACT

Operating system kernels carry a large number of security checks to validate security-sensitive variables and operations. For example, a security check should be embedded in a code to ensure that a user-supplied pointer does not point to the kernel space. Using security-checked variables is typically safe. However, in reality, security-checked variables are often subject to modification after the check. If a recheck is lacking after a modification, security issues may arise, e.g., adversaries can control the checked variable to launch critical attacks such as out-of-bound memory access or privilege escalation. We call such cases *lacking-recheck (LRC)* bugs, a subclass of TOCTTOU bugs, which have not been explored yet.

In this paper, we present the first in-depth study of LRC bugs and develop LRSan, a static analysis system that systematically detects LRC bugs in OS kernels. Using an inter-procedural analysis and multiple new techniques, LRSan first automatically identifies security checks, critical variables, and uses of the checked variables, and then reasons about whether a modification is present after a security check. A case in which a modification is present but a recheck is lacking is an LRC bug. We apply LRSan to the latest Linux kernel and evaluate the effectiveness of LRSan. LRSan reports thousands of potential LRC cases, and we have confirmed 19 new LRC bugs. We also discuss patching strategies of LRC bugs based on our study and bug-fixing experience.

CCS CONCEPTS

• **Security and privacy → Operating systems security**;

KEYWORDS

OS Kernel Bug; Missing Check; Lacking-Recheck; Error Code; TOCTTOU; Static Analysis

ACM Reference Format:
Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243734.3243844

1 INTRODUCTION

Operating system (OS) kernels, as the core of computer systems, play a critical role in managing hardware and system resources. They also provide services in the form of system calls to user-space code. In order to safely manage resources and to stop attacks from the user space, OS kernels carry a large number of security checks that validate key variables and operations. For instance, when the Linux kernel fetches a data pointer, `ptr`, from the user space for a memory write, it uses `access_ok(VERIFY_WRITE, ptr, size)` to check if both `ptr` and `ptr+size` point to the user space. If the check fails, OS kernels typically return an error code and stop executing the current function. Such a check is critical because it ensures that a memory write from the user space will not overwrite kernel data. Common security checks in OS kernels include permission checks, bound checks, return-value checks, NULL-pointer checks, etc.

A security-checked variable should not be modified before being used. Otherwise, the security check is rendered ineffective. However, in practice, due to the existence of unusual execution flows and implicit modification, a checked variable may be further modified unintentionally because of developers' ignorance of such an occurrence. If a recheck is not enforced after the modification, potential violation against the security check may occur, leading to critical security issues such as out-of-bound memory access and privilege escalation. We define such a case as an *LRC* bug—a variable is modified after being security checked, but it lacks a recheck after the modification. In other words, an LRC bug exists when two conditions are satisfied: (1) the execution has a sequence of three operations—security checking a variable, modifying the variable, and using the variable; (2) a recheck does not exist between the modification and a use of the variable.

LRC bugs can be considered a subclass of time-of-check-to-time-of-use (TOCTTOU) bugs because both of them refer to the abstract concept of modification after check but before use. However, LRC bugs differ from traditional bugs such as missing-check bugs [35, 47], double-fetch bugs [16, 33, 41, 46], and atomicity-violation bugs [14, 15, 20, 27, 42]. Inherently, LRC bugs are different from missing-check bugs [35, 47]. A check is completely absent in a missing-check bug, which by definition will not be identified as an LRC bug. For LRC bugs, checks are not "missing" at all. Existing missing-check detection [35, 47] does not warn against modifications to checked variables—so long as at least a check has been performed. We also differentiate LRC bugs from double-fetch bugs. Double fetching is a common programming practice for performance reason. It is not a bug by itself if recheck is enforced after the second copy. By contrast, LRC bugs are actual check-bypassing bugs that violate security checks. Moreover, LRC bugs target general critical data, not just the one from the user space. To compare with atomicity-violation bugs that exist in only concurrent programs, an LRC bug can exist in a single-threaded program—a thread itself may modify a security-checked variable. We will discuss in detail about the characteristics of LRC bugs by comparing them to traditional bugs in §8.

We find that LRC bugs are common in OS kernels for several reasons. First, execution paths from a security check of a variable to a use of this variable can be fairly long and complicated, especially

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243844

when involving multiple variables in both user and kernel spaces. It is extremely difficult, if not impossible, for kernel developers to reason about *all* possible execution flows on these paths, especially when multiple threads are involved. Therefore, the checked variables could be modified unintentionally when developers are not aware of such modifications. Second, harmful modifications to checked variables can occur in several common scenarios. (1) Kernel race: As the central resource manager, an OS kernel maintains many shared data structures (e.g., global variables and heap objects) for threads and processes. It is hard to ensure that various threads will never modify checked variables of those shared data structures and objects. (2) User race: Fetching data incrementally with multiple batches is common for performance reason. The security check is often performed only on the first batch, but ignored in the later batches. In practice, malicious user-space code can race to control the data in the later batches. This case shares a similar scenario to that of double-fetch bugs [41, 46]. (3) Logic errors: The thread itself may also incorrectly modify a security-checked variable due to logic errors. Such logic errors may be latent if the execution path can only be triggered with special inputs. (4) Semantic errors: Semantic errors such as type casting and integer overflow can violate a security-checked variable as well.

LRC bugs can cause critical security issues because the security checks are no longer respected. Depending on the purposes of the security checks, the security impact of LRC bugs varies. For example, if the security check is for a privilege validation, an LRC bug may succumb to a privilege escalation attack. Other common security impacts include out-of-bound memory access, arbitrary kernel memory read/write, information leaks, denial of service, etc. Therefore, it is important to detect and patch LRC bugs.

However, detecting LRC bugs in OS kernels is challenging. First, in order to detect a class of bugs, we need a clear specification of the bug patterns. LRC bugs usually result from logic errors. Unlike other bugs such as those caused by memory errors [1, 18, 37, 45], which have simple and clear patterns, LRC bugs can be complicated, and involve multiple operations (i.e., security checking a variable, modifying the security-checked variable, and using the modified variable). We need to first define and model LRC bugs so that we can automatically detect them. Second, identifying each operation constituting an LRC bug is challenging because of the lack of rules. For example, there is no general and clear rule to decide if a check is security critical or not. Third, LRC bugs typically span multiple functions. A more precise inter-procedural analysis is required. Last but not least, OS kernels are quite complex. For example, the Linux kernel has more than 22 millions lines of code and is filled with indirect calls and hand-written assemblies. The detection must not only be efficient enough to scale to millions lines of code, but also be capable of handling various challenges and corner cases.

In this paper, we first formally define LRC bugs and perform the first in-depth study of LRC bugs in the Linux kernel. We then present LRSan (Lacking-Recheck Sanitizer), an inter-procedural static analysis system for detecting LRC bugs in OS kernels. LRSan first automatically identifies security checks using a novel approach. This approach is based on an observation that if a security check fails, OS kernels will typically return an *error code*, or otherwise continue the execution. We thus define a security check as a check statement (e.g., `if` statement) that is followed by two branches, and one branch will always result in issuing of an error code while the other must have a possibility of not resulting in issuing of an error code. This way, LRSan is able to automatically infer security checks. LRSan then identifies the checked variable(s) as *critical variables*. Since the checked variables are typically derived from other variables, LRSan also employs backward analysis to recursively find the "source" variables and identifies them as critical variables as well. After that, LRSan employs a data-flow analysis to find uses of the critical variable (e.g., using a size variable in `kmalloc()`). At this point, check-use chains (e.g., execution paths) from the security checks to the uses of the critical variables are formed. In the third step, LRSan traverses these execution paths to find potential modifications of the critical variables. If a modification is found, LRSan further reasons about if a recheck is enforced between modification and use. Cases in which recheck is absent are detected as potential LRC cases. In the final step, we manually confirm LRC cases for real LRC bugs.

To the best of our knowledge, LRSan is the first system to systematically detect LRC bugs in OS kernels, which typically contain a large number (more than 131K in Linux kernel according to our recent count) of checks. Identifying security-related checks is challenging because there lacks a general rule to differentiate security checks from other checks. Finding security checks automatically thus constitutes a major building block of LRSan in LRC bug detection. We believe such security check identification is beneficial to future research in OS kernels as well. For example, identifying security checks can help us find control-dependent values, and by focusing on such values, fuzzers can significantly improve code coverage efficiently [2, 28, 29, 34]. In LRSan, we build a static global call graph and adopt a static inter-procedural analysis, which enable us to detect LRC bugs across multiple modules. LRSan's analysis is also precise, by that we mean its analysis is flow sensitive, context sensitive, and field sensitive (see §4 for details).

We have implemented LRSan based on LLVM and applied it to the Linux kernel. LRSan is able to finish the detection within four hours. The results show that LRSan find 2,808 potential LRC cases. We then manually validate those cases. At the time of paper submission, we have confirmed 19 *new* LRC bugs, most of which have been fixed with our patches by Linux developers. The results show that LRC cases are common in OS kernels and that LRSan is capable of finding LRC bugs effectively. We also perform an in-depth study on the found LRC cases and bugs. Based on the study, we summarize causes of LRC bugs, discuss bug-fixing strategies and possible improvements to the detection of LRC bugs.

Contributions. In summary, we make the following contributions.

- **A common class of OS-kernel bugs.** We identify and define LRC bugs, which can cause critical security impact such as privilege escalation or out-of-bound access. We also present the first in-depth study of LRC bugs.
- **An automated detection system.** We develop an automated system, namely LRSan, for detecting LRC bugs in OS kernels. LRSan incorporates multiple new static program analysis techniques dedicated to LRC bug detection. LRSan serves as the first step towards preventing LRC bugs. We will open source the implementation.

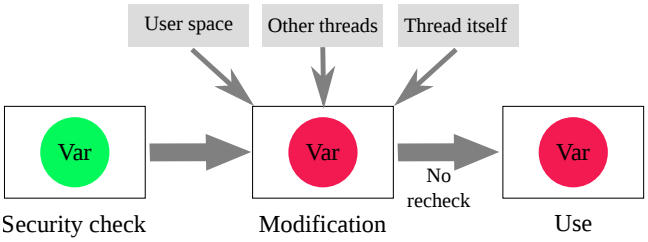


Figure 1: A sequence of three operations that form an LRC bug. The modification can come from user space, other threads, or even the thread itself. A recheck is not enforced between modification and use.

```
1  /* File: net/sctp/socket.c */
2  int min_len, max_len;
3  min_len = SCTP_DEFAULT_MINSEGMENT - af->net_header_len;
4  min_len -= af->ip_options_len(sk);
5  min_len -= sizeof(struct sctphdr) +
6       sizeof(struct sctp_data_chunk);
7  max_len = SCTP_MAX_CHUNK_LEN - sizeof(struct sctp_data_chunk);
8
9  /* val is identified as a critival variable */
10 if (val < min_len || val > max_len)
11   return -EINVAL;
12
13 if (asoc) {
14
15     /* val is modified */
16   val = asoc->pathmtu - af->net_header_len;
17   val -= af->ip_options_len(sk);
18   val -= sizeof(struct sctphdr) +
19          sctp_datachk_len(&asoc->stream);
20
21     /* val is used without a recheck */
22   asoc->user_frag = val;
23 }
```

Figure 2: An LRC example in which a checked variable is modified and used without a recheck. The code is simplified for demonstration.

- **Security check identification** We develop an automated technique to identify security checks and critical variables, which we believe can benefit related research in the future such as coverage-guided fuzzing [2, 28, 29].
- **New LRC bugs.** We have implemented LRSan based on LLVM, and applied it to the whole Linux kernel. LRSan reports thousands of potential LRC cases, and we have confirmed 19 new LRC bugs. We also discuss various strategies to fix LRC bugs.

In the rest of the paper, we present a study of LRC bugs in §2, the design of LRSan in §3, its implementation in §4, followed by an evaluation in §5. We also present some bug-fixing strategies in §6. We discuss its limitations in §7, related work in §8, and conclude the paper in §9.

2 A STUDY ON LRC BUGS

Security-checked variables should not be further modified before their uses; otherwise, they should be rechecked. LRC bugs are the cases that violate this security policy. More specifically, an LRC bug occurs when a sequence of three operations are performed on a critical variable as shown in Figure 1. The first operation checks the validity of a variable, e.g., a range check or an access-permission check. The checked variable is then modified either unintentionally or intentionally. The modification may break the security property enforced in the first operation. If the modified variable is used without being rechecked, security issues such as out-of-bound access or privilege escalation may arise.

A real LRC bug is shown in Figure 2. In this example, LRSan identifies line 10 and 11 as a security check (see §3.2) and identifies `val` as a critical variable (see §3.3). At line 16-19, `val` is modified by the thread itself. `val` is used in the later execution. However, a recheck against the value of `val` is absent, leading to an LRC bug. Note that in this example there is no validation for the variables used to redefine `val`.

LRC bugs are usually caused by semantic errors. As will be discussed in §8, LRC bugs are inherently different from traditional missing-check bugs [35, 47], double-fetch bugs [16, 33, 41, 46], or atomicity-violation bugs [14, 15, 20, 27, 42]. We now provide a formal definition of LRC bugs.

2.1 A Formal Definition of LRC Bugs

While the basic idea of finding LRC bugs is shown in Figure 1, to automatically and precisely detect LRC bugs through a program analysis, we need a formal definition of LRC bugs. An LRC bug exists with the following conditions:

- **Having a check-use chain**. Execution paths containing a security check of a variable and a use of the variable (after the check) exist. In the example of Figure 2, the check-use chain contains the execution path from line 10 to line 22.
- **Being subject to modification**. The security-checked variable might be changed in the check-use chain. Lines from 16 to 19, in Figure 2, change the value of the variable `var`.
- **Missing a recheck**. Modification to a security-checked variable is safe if a recheck is enforced after the modification. Thus, missing a recheck is also a condition to form an LRC bug.

Let us consider a program P. The variable V is in P. V has a set of possible values, $[S]$. P has a set of execution paths, $[E]$, and each execution path has a set of instructions, $[I]$. With these notations, we now define the following terms.

Security check. We define a *security check* as a conditional statement that has the following three properties. (1) It is followed by two branches (i.e., execution paths) E_t and E_c; (2) ist reads a variable V and splits its value set into V_t and V_c; and (3) E_t is control dependent on V_t, i.e., E_t is taken when the actual value of V falls in V_t, and E_c is control dependent on V_c.

In general, whether a check is a security check or not is highly dependent on developers' logic. However, based on our observation, security checks act as validators that inspect the value of a target variable, and if the value is invalid, the execution is terminated (e.g., by returning an error code). Therefore, we identify a check as a security check if E_t terminates current execution, and E_c continues the execution.

```
1  /* File: drivers/infiniband/hw/qib/qib_file_ops.c */
2  static int mmap_kvaddr(struct vm_area_struct *vma, u64 pgaddr,
3          struct qib_ctxtdata *rcd, unsigned subctxt)
4  {
5      int ret = 0;
6      ...
7          if (len > size) {
8              ret = -EINVAL;
9              goto bail;
10         }
11     ...
12 bail:
13     return ret;
14 }
```

Figure 3: An example of returning a *variable* that may contain an error code. We need data-flow to tell whether `ret` may contain an error code.

```
1  /* File: net/core/ethtool.c */
2
3  /* If FLOW_RSS was requested then user-space must be using the
4   * new definition, as FLOW_RSS is newer.
5   */
6  /* info.flow_type is identified as a critical variable */
7  if (cmd == ETHTOOL_GRXFH && info.flow_type & FLOW_RSS) {
8      info_size = sizeof(info);
9      /* info.flow_type may be modified by user race */
10   if (copy_from_user(&info, useraddr, info_size))
11     return -EFAULT;
12
13   /* Lacking-recheck: a recheck is not enforced */
14 }
15
16 /* info is used */
17 ...
```

Figure 4: A simple LRC bug caused by modification from user race.

Use. We define a use of variable V as an instruction I that takes V as a "read-only" operand for operations such as changing data and control flow.

Modification. We define a `modification` of variable V as a sequence of instructions that transform the value set of V into a new one. That is, after executing the sequence of instructions, the value set of V becomes $[V']$, and $[V'] \neq [V]$.

Lacking recheck. We define LRC as cases in which the value set of V (i.e., $[V]$) is not a subset of V_c (i.e., the valid value set right after the security check).

2.2 Security Check and Error Code in Linux

As defined in §2.1, a security check is a check that always results in a terminating execution path if the check fails. The Linux kernel incorporates a large number of security checks to validate variables and return value of functions. As the example in Figure 2 shows, if `val < min_len` or `val > max_len`, the error code `-EFAULT` will be returned, and the current execution path is terminated.

In fact, returning an error code upon a check failure is a common programming convention. Error codes are commonly used to solve the semi-predicate problem [25] in programs, i.e., to signify that a routine cannot return its normal result. In the C programming language, error handling is not provided as a language feature. Developers are responsible for detecting errors and handling them properly. As such, a programming convention for error handling is that, for functions or operations that may encounter an error or a failure at runtime, an error code is assigned to represent the result of their execution, so that the program can know if something went wrong and is able to handle it.

In the Linux kernel, most function calls return a particular error code when an error occurs. The `errno.h` and `errno-base.h` header files define 133 macros (in the latest Linux kernel) for various error codes. Figure 3 shows an example of using error codes in Linux. The error code `EINVAL` is stored in a variable. This error code indicates that `len` is a critical variable that requires a size check. We need data-flow analysis to tell whether `ret` may contain an error code upon return. A more common case is directly returning a constant error code, such as line 11 in Figure 4.

The idea in our automatic inference of a security check is that error codes shall come with security checks, i.e., an error code shall be returned if a security check fails. Security checks serve as several validation purposes: checking permission and parameters before executing a function, checking return value or output parameters after executing a function, and checking the value of a variable before using it. By identifying error codes and their associated checks, we are able to automatically find security checks.

To precisely identify security checks based on error codes, we define the patterns for identifying security checks in Linux.

- There is a conditional statement followed by two branches;
- One branch always returns an error code (*condition 1*);
- The other branch must have a possibility of not returning an error code (*condition 2*).

It is easy to understand *condition 1*—a check failure should end up with returning an error code. We define *condition 2* based on our observation that if both branches always return an error code, current conditional statement is not a security check. In this case, the actual security check is an earlier conditional statement that has already decided current conditional statement to return an error code in its both branches.

2.3 Causes of Modification

In principle, a security-checked variable should not be modified before being used. However, due to the existence of unusual execution flows and implicit modification, checked variables may be modified unintentionally, and thus opens the door for attackers to bypass security checks via exploiting such variables. In addition, OS kernels are complex. Semantic and logic errors are likely to happen, which may lead to unintended modifications to checked variables. Based on the results reported by LRSan, we study the causes of modifications and classify them into four categories. While the first two categories are from passive causes—modifications are issued by other threads through race conditions, the other two categories are by active causes—the thread itself issues the modifications due to semantic or logic errors.

User race. Kernel space frequently exchanges data with user space. Data resides in user space can be brought into kernel space through functions such as `copy_from_user()` and `get_user()`. Such data can influence the control flow and data flow, or be used for critical operations such as memory allocation. Thus, user-supplied data should be checked. To perform a security check, kernel first copies the data from user space to kernel space and then inspects its value. If the data passes the security check, further computation on the data continues. In practice, for performance reasons, kernel often

```
1  /* File: drivers/staging/media/atomisp/pci/atomisp2/atomisp_subdev.c */
2
3  /* The critical variable is the return value of atomisp_subdev_get_rect.
4   * It is an address of a shared data in "sd"
5   */
6
7  /* It is security-checked against NULL */
8  if (!atomisp_subdev_get_rect(sd, cfg, which, pad, target))
9      return -EINVAL;
10
11 /* The address of the shared data is obtained again and returned */
12
13 /* At the same time, it is used---memory read. No recheck enforced */
14  *r = *atomisp_subdev_get_rect(sd, cfg, which, pad, target);
```

Figure 5: An LRC example in which kernel thread can race to modify a shared data structure.

first peeks into a small portion of the data set instead of copying the whole data set. As the example shown in Figure 4, a small field `info.flow_type` is first copied in for the check. If the check succeeds, kernel copies the whole object `info` in. Otherwise, kernel need not copy the whole object, which can improve performance. A problem arises if user space races to modify the value between the two copies. In Figure 4, user-space code can modify data pointed to by `useraddr` in user space, which will further modify `info.flow_type` through line 10, violating the check at line 7. Our results show that user race is the most common cause of LRC bugs.

Kernel race. Race conditions in kernel space may also modify security-checked variables. OS kernels by their nature need to provide centralized resource management for all users at the same time. Therefore, in kernel space, data structures and resources are widely shared between kernel threads/processes.

According to how the shared data or resource is used across different threads/processes, there are two cases for kernel race to cause LRC bugs. The first case is similar to the case in user race, shared data (e.g., global variables and heap objects) may be copied and checked, and then copied again. The second copy may modify the checked value. Since the modification source is from the shared data, other threads/processes can race to inject malicious values to the shared data. Therefore, the modification may copy the attacker-controlled value into the security-checked variable, bypassing the security check. In the second case, kernel code directly security-checks shared data and later uses it without any copy. In this case, other threads can also race to modify the shared data to bypass the security check. As the example shown in Figure 5, object `sd` is a shared data structure in kernel space. `atomisp_subdev_get_rect()` returns the address of a field in `sd`, which is immediately checked against NULL. However, attackers can race to operate `sd` and force the function `atomisp_subdev_get_rect()` to return NULL, bypassing the security check and causing a NULL dereference.

Logic errors. Modification caused by kernel race and user race is `passive`. That is, the security-checked variable is modified by other threads. By contrast, the thread itself, due to logic errors, may also issue problematic modification leading to LRC issues. Our study reveals that common cases include value updating and resetting. Value updating is a case in which a security-checked variable is updated (e.g., through arithmetic operations) along the execution path. If the update has a logic error, it can cause an invalid value leading to a check bypass. Value resetting is common in kernel. For example, based on the type of an object, a security-checked value may be reset to a corresponding value. Example in Figure 2 shows that when `asoc` is not NULL, `val` is set to the value corresponding to `asoc`, which may violate the security check in line 10.

Semantic errors. Semantic errors such as type casting (e.g., casting a 4-byte integer to an 8-byte integer [44]) and integer overflow [6, 43] can also cause modification to a security-checked variable. If such a semantic error can only be triggered with special inputs on particular program paths, it will be hard to expose the error during normal execution. We have not yet found any LRC bug caused by such semantic errors. However, we believe it is possible given the prevalence of such semantic errors in OS kernels.

2.3.1 Lacking rechecks. We do find evidence that kernel developers enforce rechecks after potential modification. However, modification is often latent, e.g., caused by logic semantic errors, and developers may not be fully aware of potential race conditions. Rechecks thus are often missed in practice, as confirmed by the detection results (see §5.1).

2.4 Security Impact of LRC Bugs

The security impact of LRC bugs is clearly "security check bypass." Taking Figure 4 as an example, a malicious user can race to change `flow_type` in user space between line 7 and line 10. This allows the user to bypass the security check at line 7 and may cause information leakage because the following function `ops->get_rxnfc()` (omitted in the figure) will prepare various data according to `flow_type`, and copy the data to user space. This bug has been fixed in latest Linux kernel. Since the associated security check is rendered ineffective, LRC bugs can potentially have a serious security impact. This is especially true if the kernel code that has the bug is reachable by non-superusers, as is the case in Figure 4. Depending on the purpose of a security check, an LRC bug can cause various security issues. Our study reveals that common security checks include: checking permission and parameters before executing a function, checking return value or output parameters after executing a function, and checking the value of a variable before using it (e.g., pointer dereference). Correspondingly, the LRC bugs can cause privilege escalation, out-of-bound memory access, arbitrary kernel memory read/write, information leaks, denial of service, etc. Therefore, developers should be careful about potential LRC cases, and it is important for us to detect and fix LRC bugs in OS kernels.

3 LRSAN

LRSan is the first static-analysis system that aims to detect LRC bugs in OS kernels. LRSan internally incorporates a precise static-analysis engine, which employs standard data-flow and control-flow analyses. LRSan's static-analysis engine is inter-procedural, flow sensitive, context sensitive, and field sensitive. On top of the static-analysis engine, we develop multiple new analysis modules dedicated for detecting LRC bugs. We now present an overview of LRSan and the design of its key components.

3.1 Overview of LRSan

Figure 6 shows the overview of LRSan's structure and work-flow. LRSan takes as input the compiler intermediate representation (IR),

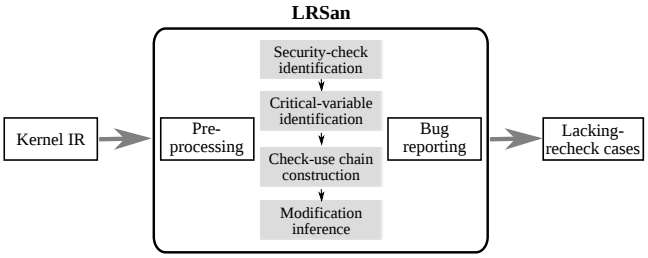
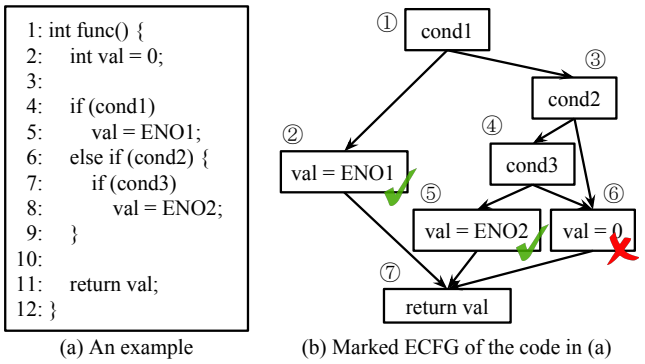


Figure 6: The structure and work-flow of LRSan.



(a) An example (b) Marked ECFG of the code in (a)

Figure 7: A simplified example of constructing ECFG.

i.e., LLVM IR, compiled from the kernel source code and automatically detects and reports LRC cases. LRSan's preprocessing constructs a global call-graph for inter-procedural analysis, and annotates error codes (macros) so that it can recognize them in LLVM IR. Specifically, LRSan includes four key components: (1) security-check identification; (2) critical-variable inference; (3) check-use chain construction; and (4) modification inference.

LRSan first employs error code–based inference (see §3.2) to automatically identify security checks. Variables associated with the security checks are then recursively identified as critical variables (see §3.3). After identifying the critical variables, LRSan identifies check-use chains (see §3.4) by taint-tracking critical variables starting from the security check. A check-use chain is formed once a use of a critical variable is found, e.g., being used as an array index or a function parameter. By constructing the check-use chains, LRSan significantly reduces its analysis scope, which allows LRSan to perform precise and expensive analysis to find LRC cases along check-use chains. LRSan's current design employs a static data-flow analysis to inspect if a security-checked critical variable is modified (see §3.5), i.e., updated with other values or overwritten through memory-related functions such as `memcpy()` and `copy_from_user()`. If a modification is found, LRSan also inspects if the security check is re-enforced after the modification to the critical variable. LRSan identifies cases without a recheck as LRC cases. LRSan's bug reporting module filters out common false positives, and reports LRC cases in a pretty format for manual confirmation. Next, we describe the design of each component in detail.

3.2 Automated Security Check Identification

Given LLVM IR, the first step of LRSan is to identify security checks. As defined in §2.2, a security check is a conditional statement (e.g., `if` statement) followed by two branches that satisfy two conditions: (1) one branch always returns an error code in the end; (2) the other branch must have a possibility of not returning any error code. At a high level, LRSan identifies security checks as follows. LRSan first collects all conditional statements (e.g., `if` statements). Then, LRSan collects all error codes and constructs an error-code CFG (ECFG) whose nodes are augmented with information on how they operate on variables to be returned, e.g., assigning an error code to such a variable. ECFG helps LRSan quickly figure out if an execution path will return an error code. With the collected information and ECFG, LRSan checks if a conditional statement is a security check by analyzing whether it satisfies the two conditions required to form a security check or not. A formal description of the work flow is shown in Algorithm 1. We now elaborate our design for security check identification.

Constructing error-code CFG (ECFG). LRSan identifies security checks by analyzing if a conditional statement satisfies the two security check forming conditions. To facilitate such an analysis, LRSan first constructs an ECFG. The motivation of building an ECFG is that the analysis procedure requires frequent query on whether an execution path (in the case that the return value is a variable rather than a constant) will return an error code or not, which requires an expensive data-flow analysis. LRSan thus first *summarizes return value–related operations* for each node in the CFG. The summary includes information such as if the node assigns an error code to a to-be-returned variable. With such a summary, the following analysis becomes efficient because the expensive data-flow analysis can be avoided.

Specifically, LRSan employs a standard *backward* data-flow analysis starting from a return instruction to find nodes in the CFG that operate return values. That is, LRSan identifies all nodes that assign a value to a to-be-returned variable. Further, for *each* to-be-returned variable in a node, LRSan marks the node as "Yes" if it assigns an error code to the variable; otherwise, it marks the node as "No." After the construction, each node in ECFG contains a list of pairs with the form `<to-be-returned variable, Yes|No>`.

Figure 7 shows an example of a function returning an error code as a variable and how to construct the corresponding ECFG. In this example, the to-be-returned variable is assigned with an error code in nodes ② and ⑤. Thus, the corresponding nodes are marked as "Yes." By contrast, the to-be-returned value is assigned with 0 in node ⑥, so it is marked as "No." All other nodes in the ECFG are not marked because they do not decide any return value. Note that, although uncommon, it is possible that a node modifies multiple return values, and thus has multiple "Yes" or "No" marks.

Identifying security checks. With the ECFG, LRSan continues the analysis to identify security checks. To achieve this, LRSan needs to first collect possible execution paths starting from the conditional statement to a return instruction. Based on the ECFG, LRSan can quickly determine whether an execution path ends up returning an error code or not. Collecting execution paths is straightforward by traversing the ECFG if the target program does not have loops. For programs with loops, unrolling is necessary. To simplify

Algorithm 1: Identifying Security Checks

Input: The marked error-code CFG $ECFG = (N, E)$
Output: The set of identified security checks $SCSet$

```
1  SCSet ← ∅;
2  MSet ← Collect_Nodes_Assigning_Return_Value (ECFG);
3  NSet ← Collect_Nodes_With_Multiple_Outgoing_Edges (ECFG);
4  for node ∈ NSet do
5      cond1 ← false; // Always return an ERRNO
6      cond2 ← false; // Possible to not return any ERRNO
7      for edge ∈ Get_Outgoing_Edges (node) do
8          PSet ← Collect_Paths_Starting_From_Edge (edge, ECFG);
9          p_errno ← true;
10         for path ∈ PSet do
11             if not Path_Return_ERRNO (path, MSet) then
12                 p_errno ← false; break;
13             end
14         end
15         if p_errno then cond1 ← true ;
16         else cond2 ← true ;
17     end
18     if cond1 and cond2 then
19         check ← Extract_Check (node);
20         SCSet ← SCSet ⋃ {check};
21     end
22 end
```

the design, LRSan chooses to unroll loops only once because the number of unrolling times typically does not affect return value. For loops with multiple paths inside the loop body, LRSan unrolls the loop multiple times to cover each of the paths with all possible orders.

Algorithm 1 shows the details of how LRSan identifies security checks. For each node that has multiple outgoing edges in ECFG, LRSan iterates all outgoing edges to check whether the two conditions of a security check are satisfied. To reason about whether an error code is returned on a specific execution path starting from an outgoing edge, LRSan only needs to check the last node that is marked for the variable to be returned on this execution path. If there is no such marked node in this path, it implies that no error code will be returned.

Now, let us apply the above algorithm to the example shown in Figure 7. Three nodes in the ECFG have two outgoing edges: ①, ③, and ④, which respectively corresponds to the if statements at line 4, 6, and 7. For ①, the path starting from the left branch always returns an error code: ① → ② → ⑦, as ② is marked as "Yes." For the other branch, one path will not return any error code: ① → ③ → ⑥ → ⑦, as ⑥ is marked as "No." Therefore, *cond1* is identified as a security check. *cond3* can also be identified as a security check in the same way. By contrast, *cond2* is not a security check because both branches may not return an error code: ③ → ④ → ⑥ → ⑦ and ③ → ⑥ → ⑦.

3.3 Recursive Critical Variable Inference

With the identified security check set (*SCSet*), LRSan then infers critical variables associated with each security check $SC \in SCSet$.

```
1  /* File: drivers/virt/vboxguest/vboxguest_linux.c */
2  struct vbg_ioctl_hdr hdr;
3  void *buf;
4
5  if (copy_from_user(&hdr, (void *)arg, sizeof(hdr)))
6    return -EFAULT;
7
8  /*
9   * hdr is identified as a critical variable; *arg is thus also
10  * recursively identified as a critival variable
11  */
12 if (hdr.size_in < sizeof(hdr) ||
13     (hdr.size_out && hdr.size_out < sizeof(hdr)))
14   return -EINVAL;
15 ...
16 buf = kmalloc(size, GFP_KERNEL);
17
18 /* *arg is in user space thus can be modified; *buf is identified
19  * as critical because it is propagated from *arg
20  */
21 if (copy_from_user(buf, (void *)arg, hdr.size_in)) {
22   ret = -EFAULT;
23   goto out;
24 }
25 ...
26 /* Lacking-recheck bug: *buf may have check-violating values */
27 ret = vbg_core_ioctl(session, req, buf);
```

Figure 8: An LRC example in which a critical variable (`*buf`) is identified through LRSan's recursive inference.

Here, being "critical" means a modification to these variables can potentially invalidate *SC*, which may lead to an LRC bug if a recheck is not enforced. One design goal in LRSan is to reduce false negatives. Therefore, LRSan tries to infer as many critical variables as possible for *SC*. Specifically, LRSan employs a standard but precise *backward* taint analysis to gather critical variables. The analysis starts with a critical-variable set *CSet*, which is initialized with variables directly used in *SC*. It then recursively includes their parent variables that propagate to the current critical variables into *CSet*.

LRSan's design of recursive critical-variable inference is also motivated by a common LRC case, i.e., the modified variable is different from the security-checked one. However, both are propagated from the same parent variable. Figure 8 shows such a case. `hdr` is first identified as a critical variable. Then, `*arg` is also identified as a critical variable because it is the propagation source of `hdr`. At the end, `*buf` is identified as a critical variable as well because it is propagated from `*arg`. With such a recursive inference, LRSan is able to comprehensively recognize critical variables.

Terminating conditions of the inference. As mentioned above, LRSan recursively collects as many critical variables as possible for the given *SC*. While being conservative to include more possible critical variables helps reduce false negatives, it may significantly increase false positives if the backward tracking continues with undecidable cases (e.g., indirect calls). Therefore, LRSan will stop backward taint tracking when it reaches hard-to-track values such as global variables, heap objects, or user-space objects. These values may come from external source or shared data, rendering further tracking difficult. However, LRSan will include them in *CSet*.

3.4 Check-Use Chain Construction

In this step, LRSan is given a pair (*SC*, *CSet*), where *SC* is a security check and *CSet* is the critical-variable set associated with *SC*. The task of this step is to construct check-use chains for each critical variable $CV \in CSet$. A check-use chain is defined as a quadruple:

<*SC*, *CV*, I_u, *PSet*>, where I_u is the first use of *CV* after *SC*, and *PSet* is a set of execution paths that start from *SC* and end with I_u. By constructing check-use chains, LRSan can delimit the actual analysis scopes in detecting LRC bugs. This allows LRSan to use a precise and expensive static analysis to detect LRC cases. To construct such a check-use chain, LRSan firstly leverages a taint-tracking analysis to find I_u and then traverses the CFG to collect the set of execution paths that start from *SC* and end with I_u. The set of execution paths is denoted by *PSet*.

The check-use chain construction shares the same static analysis engine as the one used for security check identification (§3.2) and critical variable identification (§3.3). Besides, LRSan employs alias analysis in LLVM to catch implicit uses that use critical variables through their aliases.

3.5 Modification Analysis

With a constructed check-use chain <*SC*, *CV*, I_u, *PSet*>, the last task of LRSan is to identify potential modifications to *CV* along this check-use chain. If a modification is found and no recheck is enforced, an LRC case is detected by LRSan.

Identifying modifications. Typically, there are two common types of modifications to *CV*:

- The memory location of *CV* is rewritten with a new value by a regular store instruction;
- The memory location of *CV* is modified by a memory-related function such as `memcpy()` and `copy_from_user()`.

To identify these modifications, LRSan leverages forward inter-procedural data-flow analysis. LRSan traverses the CFG starting from the *SC* specified in the check-use chain to find potential modifications. For each store instruction encountered during the CFG traversal, LRSan checks whether the memory location used in the store instruction is aliased to the memory location of *CV* or not. If yes, this store is treated as a modification to *CV*. Similarly, for a reachable memory-related function call, LRSan checks the alias results between the destination memory location of the function call and the memory location of *CV*. If they are aliased, the memory-related function call is also considered as a modification to *CV*. Note that LRSan conservatively assumes modifications to external, global, and heap objects, even if there is no explicit modification.

Missing a recheck. When a modification is found, LRSan further analyzes the code to see whether a recheck is enforced. It is necessary to do such an analysis because it is a false positive if a recheck is enforced. We do find multiple cases in which a recheck is enforced after modification. Existing double-fetch detection [41, 46] does not consider recheck, leading to false positives. Specifically, LRSan continues to traverse the CFG from the modification using a breadth-first search to reason if the modified value is rechecked by *SC*. Cases without such recheck are detected as LRC.

4 IMPLEMENTATION

We have implemented LRSan based on the LLVM compiler framework (version 6.0.0). The implementation consists of two separated LLVM passes (total about 3K lines of code). The first pass collects and prepares the information required by the static analyses in LRSan. It includes possible target functions of an indirect call, global CFG, and alias analysis results in LLVM. The second pass, with the information provided by the first pass, performs the static analyses to detect LRC cases. Due to space limitation, the rest of this section covers only important engineering issues we encountered and our solutions.

Compiling OS kernels to LLVM IR. The Linux kernel is not fully compilable under LLVM. We instead compile the Linux source code module by module. A module is skipped if it cannot be compiled successfully. The Linux source code is compiled with the "-g -O2" options. The "-g" option is used to generate debugging information, which is used to extract the source code location to facilitate the manual investigation. The "-O2" option is the default optimization level to compile the Linux kernel. We compiled the Linux kernel with "allyesconfig", which includes 16,599 modules. We successfully generate LLVM IR for 16,593 modules. That is, only 6 modules were not complied successfully.

Constructing a global call graph. LRSan relies on the CFG constructed on LLVM IR to perform the proposed static analyses. Given that LRSan's analyses are inter-procedural, a global call graph is thus required. In our implementation, we do not statically link all IR files. Instead, we adopt the iterative pass in KINT [43] to dynamic build a global call graph. For indirect calls, we use a type-based analysis [24, 38] to identify all potential targets of an indirect call.

Alias analysis. Our implementation queries the alias analysis results through the "AliasAnalysis" class in LLVM. This class provides an interface between the clients of alias analysis information and the implementations providing it. In LLVM, there are four possible results for each alias query: "NoAlias", "MayAlias", "PartialAlias", and "MustAlias", which are quite easy to understand literally. In our initial implementation, we conservatively treated two memory locations as "not aliased" only if the query result is "NoAlias". However, the preliminary results show that massive false positives can be produced due to the inaccuracy and the conservative policy adopted by the alias analysis in LLVM. We thus consider two memory locations as aliased only if the query result is "PartialAlias" or "MustAlias". Otherwise, the two locations are considered as not aliased. This significantly cuts down the number of false positives and makes manual investigation more feasible.

Analysis accuracy. As mentioned before, LRSan is built on a precise static-analysis engine—LRSan's analysis is flow-sensitive, context-sensitive, and field-sensitive. Specifically, detecting LRC bugs must use a flow-sensitive analysis because the order of security check, modification, and use must take place in sequence. LRSan constructs ECFG and detects LRC sequences by strictly following the CFG. Thus, its analysis is flow sensitive. LRSan also follows call-site specific context to facilitate a context-sensitive analysis, and leverages type information provided by LLVM to implement a field-sensitive analysis. We also manually model commonly used functions (e.g., string-related functions and some assembly code) to improve analysis efficiency and accuracy.

Recognizing error code. To identify security checks, we first need to collect error codes in LLVM IR. In Linux, error codes are defined as macros with values ranging from 1 to 133. The *lexer* preprocessor will expand error codes into constants. Identifying error codes in LLVM IR based on their values is hard. To solve this problem, we instruct the preprocessor to include error code

# SC	# CV	# CUC	# LRCC	# LRCB
131,504	86,339	99,082	2,808	19

Table 1: Detection statistics of LRSan on the Linux kernel. SC = security checks; CV = critical variables; CUC = check-use chains; LRCC = LRC cases; LRCB = found LRC bugs.

information in its metadata. With such metadata, LRSan is able to tell if a constant value is an error code or not.

Results pruning and reporting. Because modifications sometimes are a part of developers' logic, manual investigation with program semantics is required to finally confirm an LRC bug. To minimize manual effort, we need to reduce false positives without incurring too many false negatives.

From a security perspective, we heuristically prune three types of LRC cases that are likely not exploitable under existing techniques. First, if the source of a modification to a critical variable is a constant, it is unlikely to be exploitable because attackers cannot control the modification. Second, if the source of the modification to a critical variable passes the security check, it is safe to remove such LRC cases. For instance, a critical variable A is compared to another critical variable B, and the value of B is assigned to A if they are not equal. Obviously, there is no need to recheck the value of A because B's value is unchanged. Third, a mutex-style check is extensively used in OS kernels to ensure that the current state of a device or an object is updatable before changing it to a new state. Such a state change is intended, and the check is no longer effective after the change, so such LRC cases are false positives. Therefore, we need to filter out such LRC cases. We observe that error code `EBUSY` is commonly used for mutex-style check. In current implementation, we exclude `EBUSY` in our security check identification to filter out such false positives.

To facilitate manual investigation, LRSan reports the following information for each detected LRC case: the identified security check, the associated critical variable, the use of the critical variable after the security check, and the modification between the security check and the use. Given that LRSan works on LLVM IR, all of these information is reported in the form of instructions at the IR level. LRSan also reports the source code location of the related IR.

5 EVALUATION

In this section, we apply LRSan to the LLVM IR compiled from the Linux kernel source code (version 4.17) to evaluate its effectiveness and efficiency.

5.1 Detection Statistics

Table 1 shows the overall detection statistics of LRSan. In summary, LRSan identifies 131K security checks. This demonstrates that security checks are pervasive in the Linux kernel. It makes sense that OS kernels, as the core of a computer system, employ massive security checks to ensure reliability and security. Based on the identified security checks, LRSan identifies 86K critical variables after pruning. The reason why the number of critical variables is less than the number of security checks is that a critical variable may propagate to other variables, requiring multiple security checks. This number demonstrates that critical variables are prevalent, requiring extensive security checks.

Table 1 also shows that LRSan finds 99K check-use chains for identified security checks and critical variables. It is worth noting that a security check and the checked critical variable can form multiple check-use chains because the critical variable may have multiple uses after the security check. Finally, with the check-use chains, LRSan reports 2,808 LRC cases. We then manually verify these LRC cases for real LRC bugs. At the time of paper submission, our manual analysis have confirmed 19 new bugs, which are not reported by any previous bug-detection tools.

These confirmed bugs are relatively simple and thus are more obvious. There are many complicated ones, especially the ones with modification from kernel race or the thread itself, which often involve shared data and indirect calls that requires further manual investigation. It is expected that more bugs will be confirmed as our analysis continues. Indeed, as a static analysis system, LRSan has significant false positives. We will discuss how to reduce false positives in §7. LRSan is the first system towards detecting LRC bugs. We expect more improvements in the follow-up work.

5.2 Bug Finding

An important task of LRSan is to find real LRC bugs in OS kernels. Among the reported LRC cases, we have manually confirmed 19 new bugs. Table 2 shows the details and the statuses of these bugs. We have reported all of these bugs to Linux kernel developers. Twelve bugs have been fixed with our patches. Four bugs have been confirmed by Linux kernel maintainers, and we are working with them to finalize the patches. Three bugs are still pending for review and confirmation. Linux kernel maintainers would not fix two of the four confirmed bugs. For one bug, they believed that it would not cause security issues thus chose to not fix it for now. For the other one, it is in a driver that is to be removed in the near future. Therefore, they chose not to fix it. Most of the LRC bugs found by LRSan are in drivers. However, LRSan indeed finds four LRC bugs even in the well-tested core kernel.

Table 2 also shows the source of the modification in each found bug. 14 bugs have modifications coming from user space, and five bugs have modifications coming from other threads in the kernel space or the thread itself. These five bugs cannot be detected by double-fetch tools. It is worth noting that, compared to bugs with modification from user space (e.g., through `copy_from_user()`), bugs with modification from other threads in kernel space or the thread itself are much more complicated, often involving arithmetic operations, shared data among threads, and indirect calls. Confirming and fixing such bugs require more manual effort.

In addition, Table 2 shows the latent period of each bug. The average latent period of these bugs is around 5 years, which aligns with the previous finding in security flaws in the Linux kernel, i.e., the average time between introduction and fix of a kernel vulnerability is about 5 years [5]. Some of these bugs even existed for more than 10 years. This demonstrates that LRC bugs can be long-existing semantic bugs and hard-to-find. This also shows that LRSan is effective in finding long-existing LRC bugs in OS kernels.

ID	File	Function	Critical Variable	Security Check	Modi.	Y.	Status
1	drivers/.../chtls_main.c	do_chtls_setsockopt	tmp_crypto_info.version	tmp_crypto_info.version != TLS_1_2_VERSION	U	1	A
2	drivers/.../chtls_main.c	do_chtls_setsockopt	tmp_crypto_info.cipher_type	switch (tmp_crypto_info.cipher_type)	U	1	A
3	drivers/.../i2c-core-smbus.c	i2c_smbus_xfer_emulated	data->block[0]	data->block[0] > I2C_SMBUS_BLOCK_MAX	T&I	1	C
4	drivers/.../i2c-core-smbus.c	i2c_smbus_xfer_emulated	status	status != num	I	1	A
5	drivers/.../divasmain.c drivers/.../diva.c	divas_write diva_xdi_open_adapter diva_xdi_write	msg.adapter	a->controller == (int)msg.adapter	U	>10	A
6	drivers/scsi/3w-9xxx.c	twa_chrdev_ioctl	driver_command.buffer_length	driver_command.buffer_length > TW_MAX_SECTORS * 2048	U	2	A
7	drivers/scsi/3w-sas.c	twl_chrdev_ioctl	driver_command.buffer_length	driver_command.buffer_length > TW_MAX_SECTORS * 2048	U	9	S
8	drivers/scsi/3w-xxxx.c	tw_chrdev_ioctl	data_buffer_length	data_buffer_length > TW_MAX_IOCTL_SECTORS * 512	U	>10	A
9	drivers/scsi/.../mpt3sas_ctl.c	_ctl_ioctl_main	ioctl_header.ioc_number	_ctl_verify_adapter(ioctl_header.ioc_number)	U	6	S
10	drivers/scsi/sg.c	sg_write	input_size	input_size < 0	U	>10	C
11	drivers/.../llite/dir.c	ll_dir_ioctl	lumv3.lmm_magic	lumv3.lmm_magic != LOV_USER_MAGIC_V3	U	5	A
12	drivers/.../atomisp_subdev.c	atomisp_subdev_set_selection	atomisp_subdev_get_rect	!atomisp_subdev_get_rect(sd, cfg, which, pad, target)	T&I	1	C
13	drivers/.../vboxguest_linux.c	vbg_misc_device_ioctl	hdr.version	hdr.version != VBG_IOCTL_HDR_VERSION	U	1	A
14	drivers/.../vboxguest_linux.c	vbg_misc_device_ioctl	hdr.size_in	hdr.size_in < sizeof(hdr)	U	1	A
15	drivers/.../vboxguest_linux.c	vbg_misc_device_ioctl	hdr.size_out	hdr.size_out && hdr.size_out < sizeof(hdr)	U	1	A
16	net/core/ethtool.c	ethtool_get_rxnfc	info.flow_type	!(info.flow_type & FLOW_RSS)	U	1	A
17	net/sctp/socket.c	sctp_setsockopt_maxseg	val	val < min_len || val > max_len	T&I	10	S
18	net/tipc/link.c	tipc_link_xmit	imp	l->backlog[imp].len >= l->backlog[imp].limit	I	3	C
19	sound/core/control_compat.c	snd_ctl_elem_add_compat	data->type	switch(data->type)	U	>10	A

Table 2: A list of new LRC bugs detected by LRSan. `Modi.` represents modification source, in which: U is user race, T is kernel race, and I is the thread itself. `Y.` indicates the latent period in years. In the `Status` column, S is submitted, C is confirmed, and A is patch applied.

5.3 Analysis Time

Our experimental platform is equipped with a Quad-Core 3.5 GHz Intel Xeon E5-1620 v4 processor. The main memory is 32 GB, and the operating system is Ubuntu 16.04 with Linux-4.4.0. LRSan finishes the analysis of the Linux kernel (with "allyesconfig") within four hours, which demonstrates the efficiency of LRSan. After investigating the analysis process, we found that more than 80% time is consumed by the first pass (i.e., information collection) of LRSan. This is reasonable because the first pass needs to iteratively build a global CFG and collect the alias-analysis results. Given that there is no dependency between these two tasks, it is possible to further reduce the analysis time by parallelizing them, as multi-core processors have been broadly available in today's machines. Alternatively, the first pass needs to run only once, so we can save the collected information and reuse it, instead of running it every time.

6 BUG-FIXING STRATEGIES

According to LRSan's detection results, LRC bugs are common in OS kernels. To avoid security issues, an LRC bug should be fixed immediately once it is exposed. Based on our study and experience in LRC bugs and our communication with Linux kernel maintainers regarding patching the bugs found by LRSan, we summarize possible bug-fixing strategies against LRC bugs as follows.

Enforcing a recheck. An intuitive fixing strategy is to enforce a recheck after the identified modification. For example, to fix the LRC bug in Figure 2, we can re-enforce the security check (line 10-11) right after the modification (line 16-19). Such a fixing strategy is straightforward and particularly effective to cases in which a checked variable needs to be intentionally updated after the check, e.g., modification also happens in normal execution. However, such a fix is to detect the potential security bleach from the modification, but not to prevent a harmful modification—the root cause of the bug. The shortcomings of this fixing strategy include: (1) introducing extra performance overhead; (2) duplicating checks can cause cleanness concerns, due to which, Linux kernel maintainers may become reluctant to apply the patch.

Caching checked variable. Another fixing strategy is to render the modification ineffective by caching and reusing the checked variable. After a security check, we can temporarily cache the checked value in a separate variable. Later on, to prevent the security impact from a harmful modification, we have two choices: either immediately overwriting the potentially modified variable with the cached value or using the cached value whenever the variable is used. Such a fixing strategy is not applicable to cases in which a security-checked variable needs to be intentionally updated (e.g., increasing an index variable of an array). Moreover, if the security check and the following uses of a critical variable cross multiple functions, especially when indirect calls are involved, such a fix becomes complicated and requires deep understanding of the code.

Avoiding harmful modifications. A more fundamental bug-fixing strategy is to avoid harmful modifications. For example, in Figure 4, we can copy the whole `info` object in the first place to avoid the second copy. Such a fixing strategy works for cases in which there is no need to update the checked variable. Such a fix can be expensive, especially when copying a large object, and complicated if refactoring the code is required to avoid the modification.

Using transactional memory. An alternative to prevent harmful modifications is to use transactional memory, which ensures a

	False Positive Cause	Description	Percentage	Elimination Technique
1	Checked modification	Value used for modification has been checked before	25%	Symbolic execution
2	Satisfiable modification	Modification satisfies the security check, e.g., assigning statically-known values	20%	Symbolic execution
3	Uncontrollable modification	Modification is uncontrollable to attackers, e.g., internal kernel variables	38%	Taint analysis
4	Transient check	Security check expires because the modified variable is redefined for other purposes	6%	N/A
5	Unconfirmed race	Source variable of modification may have kernel races, but is hard to confirm manually	9%	Race detection
6	Other	Inaccurate global call graph, unreachable code, etc	2%	Analysis optimization

Table 3: A study of causes of false positives in the detection results of LRSan. Except false positives caused by transient checks, others can be eliminated or reduced through automatic techniques such as symbolic execution.

group of load and store instructions are executed in an atomic way. DECAF [33] uses Intel transactional synchronization extensions (Intel TSX) [12] to implement this fixing strategy, i.e., a transaction will abort if a modification to the protected variable from other threads occurs. Such a fixing strategy is expensive, often requires hardware support and thus only works for special processors.

7 LIMITATIONS AND DISCUSSION

While LRSan has demonstrated the capability to find a large number of LRC cases and expose real LRC bugs in the Linux kernel, LRSan does have its own limitations. In this section, we discuss these limitations as well as potential solutions. We will also discuss about how to extend LRSan to detect LRC bugs in other software systems such as web servers and database systems.

7.1 False Positives

As a static analysis system, LRSan inevitably suffers from false positives. LRSan identifies 2,808 LRC cases for the entire Linux kernel, which consists of 22+ millions source lines of code. For the core Linux kernel, LRSan detects only 340 LRC cases. To filter out false positives, we manually analyzed LRC cases detected by LRSan, which took two researchers about 40 hours in total. We believe this effort is moderate and manageable. Our manual analysis considers an LRC case a false positive if it is safe to perform the modification. For example, the source value used for the modification may be checked before, and thus the modification is valid. Note that LRC cases that are hard to manually confirm are also conservatively treated as false positives. Table 3 summarizes our study of false positives. We discuss each category in detail as follows.

Checked modification. Our manual investigation reveals that many false positives stem from checked modifications in which the source of the modification is already checked; therefore, the modification is safe and would not cause an LRC bug. Such false positives can be eliminated through symbolic execution. By symbolically computing the value of the source of modification, we can automatically verify whether the modified value will satisfy the security check.

Satisfiable modification. Another common cause of false positives is satisfiable modifications. For example, a range-checked `index` is incrementally updated in the computation on a *list* structure. Such a modification (i.e., updating) is valid as long as its updated value still satisfies the security check, which is usually enforced the next time when `index` is used. Similar to the case of checked modification, we can also automatically eliminate false positives caused by satisfiable modification through symbolic execution—by symbolically computing the value of the critical variable and verifying whether it still satisfies its security check.

Uncontrollable modification. LRC cases in which the source variable of a modification is uncontrollable to attackers are deemed as false positives because they are not exploitable. For example, the value of a modification may come from an internal kernel variable. To eliminate such false positives, existing techniques such as taint analysis track the source of a modification and figure out whether it is controllable to the external world. Given that an LRC bug in the kernel may be invoked only by a superuser and not by an unprivileged user, which depends on the access control policy of the target system, inferring exploitability may also require the knowledge of access control.

Transient check. We also find a few false positives caused by transient security checks. That is, a security check has a limited liveness scope, and a modification occurs outside the liveness scope of the security check. In other words, a variable is redefined (i.e., modified) for other purposes, and thus the previous security check expires. Therefore, further modification to the variable may not violate the previous security check. Automatically understanding the *purpose* of a variable is hard because it is highly dependent on program developers' logic. Given that such cases are not common, we leave the filtering for manual analysis.

Unconfirmed race. If the source variable of a modification is from a shared variable (e.g., globals), it is extremely hard, if not impossible, for manual analysis to figure out its possible values without analyzing potential race conditions. As a result, we conservatively classify such LRC cases as false positives. In the future, we plan to equip LRSan with existing race detection techniques to understand how the shared variable can be controlled by other threads.

Other. Some false positives are caused by typical limitations with static analysis, e.g., inaccurate global call graph. While they are orthogonal challenges, analysis optimization techniques such as pointer analysis [10, 11] for finding targets of indirect calls can further improve the analysis accuracy of LRSan.

7.2 False Negatives

LRSan also has potential false negatives. In current design, LRSan leverages error codes to infer security checks because a majority of security checks return an error code upon failures. However, such an approach may miss some cases where no error code is returned upon a failed security check. For example, if a `size` value is larger than a default maximum number, the kernel code may choose to reset its value to the default maximum number and does not return any error code. Our current implementation adopts "MustAlias" (in contrast to "MayAlias") results in the taint tracking, which can also cause false negatives when actual aliases are missed. Moreover, we do not include kernel modules that cannot be successfully compiled

by LLVM. We manually modeled only commonly used assemblies but not all. These issues may also cause false negatives.

7.3 Supporting More Systems

The techniques developed in LRSan are not limited to LRC bugs in the Linux kernel. In fact, the automated security-check identification and recursive critical-variable inference do not have specific assumptions on the target bugs or programs. The only assumption LRSan makes is the availability of error codes. Some higher-level programming languages tend to use other error-handling mechanisms. However, the concept of error code is general. For example, in the C++ programming language, in addition to error codes, exceptions are also a common mechanism to handle errors. Similar to error codes, developers define various exception handlers to handle different kinds of exceptions. Moreover, widely-used system software such as OS kernels, web servers, and database systems all have error code–like mechanisms to handle errors and failures. By specifying how to recognize such "error code," we can naturally extend LRSan to detect LRC bugs in other software systems.

8 RELATED WORK

LRSan employs static program analysis techniques to detect LRC bugs in OS kernels. In this section, we identify differences between LRSan and some related work that leverages kernel code analysis techniques to find semantic bugs including missing-check bugs, double-fetch bugs, and atomicity-violating bugs.

8.1 Kernel Code Analysis

OS kernels are attractive attack targets. A single vulnerability in an OS kernel can have a critical and wide-ranging security impact. Recent research on kernel code analysis has been focusing on improving practicality, accuracy, and soundness.

K-Miner [8] is a new static analysis framework for OS kernels. Its key idea is to partition the kernel code along separate execution paths using the system call interface as a starting point. This way, it significantly reduces the number of relevant paths, allowing practical inter-procedural data-flow analysis in complex OS kernels. K-Miner is able to detect memory-corruption vulnerabilities such as use-after-free. Dr. Checker [21] is also a static analysis tool for identifying bugs in Linux kernel drivers. It reduces analysis scope by focusing only on drivers, and improves scalability and precision by sacrificing soundness in a few cases such as not following calls into core kernel code. It is able to effectively detect known classes (e.g., uninitialized data leaks) of vulnerabilities in drivers. Both K-Miner [8] and Dr.Checker [21] aim to improve practicality and precision of static kernel analysis by limiting analysis scope, and they employ traditional bug detection techniques. LRSan instead aims to detect LRC bugs, a specific class of semantic errors in OS kernels, which has not been explored before, and to this end, LRSan incorporates new analysis techniques such as automated security check identification and recursive critical variable inference. Moreover, LRSan's analysis scope covers the whole OS kernel, including both drivers and core kernel modules.

KINT [43] and UniSan [19] both are capable of statically analyzing whole OS kernels. KINT [43] uses taint analysis to find integer errors in the Linux kernel while UniSan [19] uses taint analysis to find uninitialized bytes that may be copied from kernel space to user space. Smatch [3] employs Sparse [39] to construct syntax tree with type and structure information. It provides lightweight, intra-procedural static analysis to find shallow bugs such as NULL pointers and uninitialized uses. Coccinelle [26] is a pattern-matching and transformation system for OS kernels. It uses a dedicated language SmPL (Semantic Patch Language) to specify patterns and transformations in C code. Coccinelle is highly scalable because it does not perform expensive inter-procedural data-flow analysis. Compared to these analysis tools, LRSan employs whole-kernel inter-procedural data-flow analysis, which is flow sensitive, context sensitive, and field sensitive. In addition, we design multiple new analysis techniques in LRSan to detect LRC bugs.

Symbolic execution [17] "executes" programs using symbolic values. As well as solving the imprecision in traditional static analysis, it can cover significantly more execution paths of a program than dynamic analysis. With the substantial improvement in the efficiency of symbolic execution, e.g., S2E [4], recent research has been able to symbolically execute OS kernels. S2E is a platform used to analyze properties and behaviors of software systems like OS kernels. It enables selective symbolic execution to automatically minimize the amount of code that has to be executed symbolically, and relaxed execution consistency models to trade-off performance and accuracy. APISan [48] detects API misuses by analyzing function usage patterns. It utilizes symbolic execution to reason about program errors caused by API misuses. SymDrive [30] also uses symbolic execution to verify properties of kernel drivers. LRSan detects LRC bugs with a principled and general approach. Symbolic execution, an orthogonal technique, can improve the analysis precision of LRSan. For example, symbolic execution can precisely reason about whether a modified value still satisfies the constraints of the security check, which is an interesting research issue.

8.2 Missing-Check Bugs

Security checks validate inputs and operation status to ensure security. Missing check, in which a check is completely absent, is a very related class of bug as LRC. However, they are inherently different. By definition, cases in which a check is completely missed are not LRC bugs. On the other hand, in LRC, a security check is actually present for a variable, and thus, it is not missing-check.

As the first step of detecting missing-check bugs, it is necessary to find some evidence that a security check is required for a case. While manual auditing is feasible, it is definitely not scalable. In order to automatically find such evidence, researchers have employed static analysis to infer missing-check cases. A previous approach [7] collects sets of programmer beliefs, which are then checked for contradictions to detect various types of missing-check bugs. JIGSAW [40] automatically constructs programmer expectation on adversarial control at a particular resource access to detect missing checks. Chucky [47] uses check deviations to infer missing check. For example, if a length variable is checked in 9 out of 10 functions in a program, it is evident that the last function may miss the check. Similarly, Juxta [22] automatically infers high-level semantics by comparing multiple existing implementations that obey latent yet implicit high-level semantics. Deviations are detected as

potential missing checks. RoleCast [35] exploits universal and common software engineering patterns to infer missing checks in web applications. MACE [23] identifies horizontal privilege escalation (HP) vulnerabilities in web applications using program analysis to check for authorization state consistency.

In addition to focusing on a common class of semantic bugs, LRSan differs from aforementioned missing-check detectors in that it can identify security checks without the requirements of multiple implementations of target systems or manual specifications.

8.3 Double-Fetch Bugs

Recent research has attempted to detect a class of semantic bugs named double-fetch in OS kernels. A double-fetch bug is a case in which same user-space data is fetched into kernel space twice. Since malicious users may race to change the data between the two fetches, inconsistent data could be fetched. A double fetch itself is not a security issue but rather just improper programming practice. Depending on how the fetched data is used and checked, a double-fetch case may become a security bug. The Bochspwn project [16] introduced double-fetch bugs for the Windows kernel. Wang et al. [41] then systematically studied double fetches and employed patch matching to find successive fetches (e.g., `copy_from_user()`). DECAF [33] exposes double fetches through cache side channel information because double fetches leave cache patterns. All these detectors do not further check if a double-fetch would lead to security issues, causing significant false positives. Deadline [46] improves double-fetch detection and detects double-fetch bugs by checking whether the value may influence control flow or data flow between the fetches. However, if the value obtained from the second fetch is rechecked or never used for security-related operations, such a double fetch is still not a real bug. By contrast, LRSan aims to detect actual check-bypassing bugs. That is, a critical variable with check-violating values is used. Moreover, LRSan targets general critical data (i.e., not just the one from user space) and general modification (i.e., not just from `copy_from_user()`). Five new bugs found by LRSan target non-user data, which by definition cannot be identified by double-fetch detection.

8.4 Atomicity Bugs

When a modification is coming from another thread, LRC has an overlap with atomicity violation (by contrast, double-fetch is a class of race condition, not atomicity violation). Atomicity is a generic concurrency property that ensures proper synchronization on accesses from multiple threads to shared data structures. Atomicity violation has been extensively studied in previous work [14, 15, 20, 27, 42]. In high level, existing detection tools define atomicity-violation bugs as cases in which a variable is shared by multiple threads, and one thread has a pair of accesses to this variable while other threads may potential write to or read from the variable between the two accesses. Such a detection mechanism is not suitable for detecting LRC bugs due to several reasons: (1) the variable in LRC may not be shared but modified locally in the same thread; (2) the modification may cross user-kernel boundary or come from global or heap; and (3) the traditional detection pattern is too general and thus suffers from significant false positives. In LRSan, we design multiple dedicated mechanisms to detect LRC bugs. LRSan focuses on critical variables inferred from identified security checks and is able to detect modification from local.

More importantly, if the modification is from the thread itself, an LRC bug is not an atomicity-violation bug. In other words, LRC bugs can exist in single-threaded programs. Therefore, LRC also differs from atomicity violation.

8.5 Error-Code Analysis

Prior research works have tried to detect and monitor error-code propagation [9, 13, 31, 32]. However, they mainly focused on analyzing the completeness and correctness of error-code propagation and handling, instead of leveraging error codes to infer security checks. Kenali [36] attempts to infer access controls based on error code `-EACCES`. However, it uses a simple pattern-matching approach to find `return -EACCES` and treats the closest `if` statement as the access-control check. Such an approach misses cases where an error code is assigned to a to-be-returned variable. Kenali does not support *general* error codes nor formally define a security check, suffering from significant false reports. In comparison, we formally define security checks. LRSan is capable of systematically and precisely find security checks using ECFG. LRSan is also able to recursively find the associated critical variables.

9 CONCLUSION

OS kernels validate external inputs and critical operations through a large number of security checks. LRSan identifies more than 131K security checks in current Linux kernel. A security-checked variable should not be modified before being used. Otherwise, the security check is rendered ineffective. If a recheck is missing after a modification, various critical security issues may arise. Attackers can control the security-checked variable to bypass the security check, causing attacks such as privilege escalation, out-of-bound access, or denial-of-services. We call such cases LRC bugs, a specific class of semantic errors that has not been explored before.

This paper presents the first in-depth study of LRC bugs, including the formal definition, common causes, and security impact of such bugs. This paper also describes LRSan, the first automated static analysis system for detecting LRC cases in OS kernels. LRSan is equipped with multiple novel techniques such as automated security check identification and recursive critical-variable inference to systematically detect LRC cases. LRSan's static analysis is inter-procedural, flow sensitive, context sensitive, and field sensitive. We have implemented LRSan based on LLVM and applied it to the latest Linux kernel. LRSan is able to finish the detection within four hours. Detection results show that LRSan finds 2,808 potential LRC cases. At the time of paper submission, we have found 19 new LRC bugs, most of which have been confirmed or fixed with our patches by Linux kernel maintainers. The results show that LRC is a common class of semantic errors in OS kernels and that LRSan is capable of finding LRC bugs effectively.

ACKNOWLEDGMENTS

We would like to thank Hayawardh Vijayakumar and the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported in part by the National Science Foundation under Grant No. CNS-1514444.

REFERENCES

[1] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 263–277. https://doi.org/10.1109/SP.2008.30

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[3] Dan Carpenter. 2009. Smatch - the source matcher. http://smatch.sourceforge.

[4] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[5] Kees Cook. 2017. Linux Kernel Self Protection Project. https://outflux.net/slides/2017/lss/kspp.pdf.

[6] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 760–770. http://dl.acm.org/citation.cfm?id=2337223.2337313

[7] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041

[8] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *2018 Network and Distributed System Security Symposium (NDSS '18)*.

[9] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, USA, Article 14, 16 pages. http://dl.acm.org/citation.cfm?id=1364813.1364827

[10] Ben Hardekopf and Calvin Lin. 2007. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 290–299. https://doi.org/10.1145/1250734.1250767

[11] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 289–298. http://dl.acm.org/citation.cfm?id=2190025.2190075

[12] Intel. June, 2017. Programming with Intel Transactional Synchronization Extensions. In *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Chapter 16*.

[13] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, Berkeley, CA, USA, 345–362. http://dl.acm.org/citation.cfm?id=3241094.3241122

[14] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[15] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-bug Fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 221–236. http://dl.acm.org/citation.cfm?id=2387880.2387902

[16] Mateusz Jurczyk and Gynvael Coldwind. 2013. Bochspwn: Identifying 0-days via System-wide Memory Access Pattern Analysis. (2013).

[17] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[18] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 81–96. http://dl.acm.org/citation.cfm?id=2831143.2831149

[19] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 920–932. https://doi.org/10.1145/2976749.2978366

[20] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

[21] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1007–1024. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[22] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 361–377. https://doi.org/10.1145/2815400.2815422

[23] Maliheh Monshizadeh, Prasad Naldurg, and V. N. Venkatakrishnan. 2014. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 690–701. https://doi.org/10.1145/2660267.2660337

[24] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 577–587. https://doi.org/10.1145/2594291.2594295

[25] Peter Norvig. 1992. The General Problem Solver.

[26] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. ACM, New York, NY, USA, 247–260. https://doi.org/10.1145/1352592.1352618

[27] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[28] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP '18)*, Vol. 00. 917–930. https://doi.org/10.1109/SP.2018.00056

[29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *2017 Network and Distributed System Security Symposium (NDSS '17)*.

[30] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. 2012. SymDrive: Testing Drivers Without Devices. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 279–292. http://dl.acm.org/citation.cfm?id=2387880.2387908

[31] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error Propagation Analysis for File Systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 270–280. https://doi.org/10.1145/1542476.1542506

[32] Cindy Rubio-González and Ben Liblit. 2011. Defective Error/Pointer Interactions in the Linux Kernel. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 111–121. https://doi.org/10.1145/2001420.2001434

[33] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2017. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. *CoRR* abs/1711.01254 (2017). arXiv:1711.01254 http://arxiv.org/abs/1711.01254

[34] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1492–1504. https://doi.org/10.1145/2976749.2978411

[35] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2011. RoleCast: Finding Missing Security Checks when You Do Not Know What Checks Are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 1069–1084. https://doi.org/10.1145/2048066.2048146

[36] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *2016 Network and Distributed System Security Symposium (NDSS '16)*.

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[38] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 941–955. http://dl.acm.org/citation.cfm?id=2671225.2671285

[39] Linus Torvalds. 2006. match - the source matcher. https://sparse.wikii.kernel.org/index.php/Main_Page.

[40] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. 2014. JIGSAW: Protecting Resource Access by Inferring Programmer Expectations. In *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX*

Security 14). USENIX Association, Berkeley, CA, USA, 973–988. http://dl.acm.org/citation.cfm?id=2671225.2671287

[41] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. 2017. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1–16. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei

[42] Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, and Yong Guan. 2014. Localization of Concurrency Bugs Using Shared Memory Access Pairs. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 611–622. https://doi.org/10.1145/2642937.2642972

[43] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 163–177. http://dl.acm.org/citation.cfm?id=2387880.2387897

[44] Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. 2016. Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 541–552. https://doi.org/10.1145/2976749.2978403

[45] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 529–540. https://doi.org/10.1145/2976749.2978340

[46] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[47] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*. ACM, New York, NY, USA, 499–510. https://doi.org/10.1145/2508859.2516665

[48] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 363–378. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun