Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs

Kangjie Lu, Aditya Pakki, and Qiushi Wu

University of Minnesota

Abstract. OS kernels enforce a large number of security checks to validate system states. We observe that security checks are in fact very informative in inferring critical semantics in OS kernels. Specifically, security checks can reveal (1) whether an operation or a variable is critical but can be erroneous, (2) what particular errors may occur, and (3) constraints that should be enforced for the uses of a variable or a function. Such information is particularly valuable for detecting kernel semantic bugs because the detection typically requires understanding critical semantics. However, identifying security checks is challenging due to not only the lack of clear criteria but also the diversity of security checks.

In this paper, we first systematically study security checks and propose a mostly-automated approach to identify security checks in OS kernels. Based on the information offered by the identified security checks, we then develop multiple analyzers that detect three classes of common yet critical semantic bugs in OS kernels, including NULL-pointer dereferencing, missing error handling, and double fetching. We implemented both the identification and the analyzers as LLVM passes and evaluated them using the Linux kernel and the FreeBSD kernel. Evaluation results show that our security-check identification has very low false-negative and false-positive rates. We also have found 164 new semantic bugs in both kernels, 88 of which have been fixed with our patches. The evaluation results confirm that our system can accurately identify security checks, which helps effectively identify numerous critical semantic bugs in complex OS kernels.

Keywords: OS kernel · semantic bug · security check · missing check · error handling.

1 Introduction

OS kernels not only process inputs from both arbitrary user-space programs and underlying hardware, but also manage resources and perform complicated operations that are often error-prone. To ensure the security, OS kernels carry out a large number of `if` and `switch` statements (we refer to them as *conditional statements* [33] hereafter for simplicity) to validate system states. A conditional statement becomes a security check when it is used to capture errors. That is, a security check is a conditional statement that sanitizes erroneous states, such as lines 4 and 9 in Figure 1. By contrast, normal conditional statements do not capture errors but are used to select normal execution paths, such as line 2 in Figure 1. We will present the definition of security checks in §2.1.

While security checks are intended to prevent erroneous states, they are in fact very informative in understanding critical semantics. In particular, security checks reveal at least three kinds of code semantics: (1) whether an operation or a variable is security-critical but can be erroneous, (2) what particular errors the code may have, and (3) what constraints should be enforced for the uses of a variable or a function. For example, the security check at line 4 of Figure 1 reveals that the variable addr is critical but erroneous—its value can be out of the valid boundary. Further, we can infer that addr might be used for memory accesses, and to avoid potential out-of-bound accesses, the "within-boundary" constraint should be enforced before using addr. Such information is valuable for detecting critical semantic bugs. By identifying which variables are critical but can be erroneous, we can inspect whether they are thoroughly validated before being used and whether the errors are handled properly. On the other hand, security checks themselves are problematic, e.g., a security check can be invalidated by race conditions, or a security check may target a wrong condition.

```
1  int ret = -EINVAL;
2  if (flags & VM_SHARED) {
3    page = vmf->page;
4    if (addr < vm_end)
5      ret = 0;
6  }
7  else {
8    page = vmf->cow_page;
9    if (!page_count(page))
10     // error handling
11     // or fixing
12     BUG();
13   ret = put_page(page);
14 }
15 // error-code returning
16 return ret;
```

Fig. 1: Two security checks (lines 4 and 9, but not 2).

It is worth noting that identifying security checks is also useful for fuzzing and system hardening. Fuzzing techniques [10] are often inefficient in exploring deep paths because of the barriers set by checks [3, 9, 20]. By specifically solving the constraints from security checks and focusing fuzzing on checked variables, one can efficiently guide the fuzzing to pass through the security checks and have a better chance to trigger critical issues. On the other hand, existing system-hardening techniques such as memory safety [18, 24] and fault/memory isolation [1, 11, 16, 27, 36] tend to have a high performance overhead. Identifying critical semantics would allow researchers to selectively focus on a small set of protection targets and thus improve performance.

While the identification of security checks presents rich opportunities for detecting critical semantic bugs and hardening systems, it is a challenging task. First, whether a conditional statement is a security check depends on its context. To decide if a conditional statement is a security check, one must consider multiple aspects: where the checked variable comes from, how it is checked, what it is used for, how and where it is used, what the potential reliability and security impact is, etc. All these aspects depend on the context of the code and developers' logic. Further, security checks can be highly diverse in form—we have neither clear criteria for identifying security checks nor even definition of them. Therefore, it is a challenge to automatically identify security checks.

In this paper, we aim to automate the identification of security checks in complex and large OS kernels, and use the identified security checks to detect various classes of common and critical kernel semantic bugs, including NULL-pointer dereferencing, missing error handling, and double fetching. We first carry out a study on security checks to understand their characteristics. Based on the study, we then develop CHEQ (security-check magnifier), an automated tool for identifying security checks in OS kernels. The key insight behind CHEQ is that security checks validate erroneous states, hence checking failures require *handling*. The hard problem of reasoning about the context and properties of conditional statements is therefore transformed into the one of identifying failure-handling (FH for short) code, which can be automated because FH has clear patterns. According to our study, FH patterns include (1) returning an error code, (2) stopping the current execution, (3) fixing the erroneous state (typically the checked variable), and (4) issuing error messages. We call them *FH primitives*.

After identifying security checks, we further use them to detect three classes of common and critical semantic bugs: NULL-pointer dereferencing, missing error handling, and double fetching. Specifically, NULL-pointer dereferencing is that a pointer can be NULL but is dereferenced without any NULL check. Missing error handling is that a returned error code is not handled, e.g., being ignored. Double-fetch bugs are cases in which an external variable is copied into the kernel space and security-checked; however, the buggy code copies the variable in again and uses it without a further check. As shown in prior works [2, 6–8, 12, 25, 31, 35], all these classes of bugs are common and may cause critical security issues including system crashes, data losses, information leaks, and even privilege escalation. Unlike prior works, our detection focuses on security-checked variables that can cause critical impacts, thereby eliminating overwhelming false-positive and non-critical cases, as shown in §6.2.

We implemented CHEQ and semantic-bug detection as multiple LLVM analysis passes. We then extensively evaluated both of them. Evaluation results show that CHEQ successfully identifies 97.5% ground-truth security checks we collected from patches for recent missing-check bugs and that 98.3% of identified security checks are real. These results show that CHEQ is accurate in identifying security checks. We have also applied our bug detection to both the Linux kernel and the FreeBSD kernel and found 164 new bugs. Specifically, we found 24 NULL-pointer dereferencing bugs, 128 missing error handling bugs, and 12 double-fetch bugs. We have reported the bugs in the Linux kernel to maintainers, and patches for 88 bugs have been accepted. The promising detection results mainly benefit from CHEQ's accurate identification of security checks because security checks help identify critical variables and functions that can be erroneous and reveal the constraints that should be enforced when they are used.

We make the following contributions in this paper:

- **A study of security checks.** We conduct a study of security checks to find intrinsic differences between security checks and normal conditional statements, and to classify how errors are typically handled.
- **Automatic identification of security checks.** We propose a mostly-automated approach to identify security checks in OS kernels. The approach incorporates multiple techniques such as identifying custom error codes and constructing a control-flow graph marked with FH primitives.
- **Detection of three classes of semantic bugs.** We harness CHEQ for the detection of three classes of common and critical semantic bugs: NULL-pointer dereferencing, missing error handling, and double-fetching. We have found 164 new semantic bugs in the Linux kernel and the FreeBSD kernel. Many of these bugs have been fixed with our reported patches.
- **A new, open-source tool.** We implement CHEQ as an easy-to-use LLVM pass and open-source [1] it to facilitate future research on semantic-bug detection and fuzzing.

[1] https://github.com/umnsec/cheq

2 A Study of Security Checks

OS kernels have prevalent conditional statements such as `if` and `switch` statements. The majority of conditional statements are normal *selectors* but not security checks, as will be shown in our evaluation (§6.1). In the code sample shown in Figure 1, line 2 is an example of selectors that sets `page` and continues normal execution. Lines 4 and 9 are instead security checks because they are intended to capture errors. To differentiate security checks from normal selectors, we need a definition of security checks. To this end, we investigated a set of ground-truth security checks. Previous papers and Linux maintainers reported and fixed numerous missing-check bugs. Intuitively, these bugs were typically fixed by inserting security checks. Therefore, we can collect ground-truth security checks from fixes for previous missing-check bugs. Specifically, we collected 40 security checks from previous papers [15, 31, 32, 34, 35]; 50 and 20 security checks from the `git` patch history of the Linux kernel and the FreeBSD kernel, respectively. Based on the intrinsic features of the security checks, we provide the definition of security checks, which does not adhere to any specific program. With the generic definition, we further manually collected 490 security checks from the source code of different modules in both kernels, and characterized FH primitives.

2.1 Definition of Security Checks

We first define security checks based on their intrinsic features that do not adhere to any specific programs. We observe that security checks are intended to capture errors, so their conditions (in the `if` or `switch` statements) indicate erroneous states. Once an error is captured, at least one branches will handle it, and at least one branches will continue the normal execution.

To formally define security checks, we suppose that B_C is a basic block that contains a conditional statement. B_C has multiple successor basic blocks, B_S. B_{Si}, where i is an integer index, represents the "ith" successor basic block. B_R is a basic block that contains a return instruction in the current function. We define sequence S_i as [B_C, B_{Si}, ..., B_R], which represents all code paths that start from B_C, contain B_{Si}, and end with B_R. Note that, since there could be multiple sub-paths between B_{Si} and B_R, S_i may cover multiple code paths that contain B_C, B_{Si}, and B_R. With these terms, we use S_i to represent branch "i" of B_C and define the FH property P of S_i as follows.

$$P(S_i) = \begin{cases} \text{MUST-FH if all of its following code paths have FH primitives} \\ \text{NO-FH if none of its following code paths has FH primitives} \\ \text{MAY-FH otherwise} \end{cases} \tag{1}$$

We then identify the conditional statement in B_C as a security check if:

$$\begin{aligned} \exists S_i \text{ such that } P(S_i) = \text{MUST-FH} \\ and \\ \exists S_j\text{, where } i \neq j\text{, such that } P(S_j) = \text{MAY-FH or NO-FH} \end{aligned} \tag{2}$$

In other words, we identify a conditional statement as a security check if it has at least one MUST-FH branches and at least one MAY-FH or NO-FH branches.

2.2 Handling Security-Check Failures

In this section, we study the common ways of handling checking failures.

Returning error codes. The most common strategy for handling a checking failure is to return an error code upstream, and callers will take care of handling the erroneous states based on the error codes. Error codes can be *standard* or *custom*. OS kernels usually define standard error codes in a uniform manner. For example, the Linux kernel defines 150 standard macro error codes (e.g., `EINVAL`) in `errno.h` and `errno-base.h`. In addition, `-1` and `NULL`, when used as a return value, are also standard error codes. However, `false` is often used to represent a normal Boolean value instead of an error. In subsystems, developers may define custom error codes to represent module-specific errors. For example, `VM_FAULT_*` are custom error codes used in the memory-management module. Unlike standard error codes, custom error codes are diverse and defined in different files, which makes the identification of custom error codes challenging.

Issuing error messages. In many cases, developers are aware of the failure scenarios of their code. While production code is never released with `assert()` turned on, a failure is notified via an error message. Although most kernel drivers and file systems have custom error-message functions to log failures, such functions have clear patterns. Taking the Linux kernel as an example, such functions often have a name ending with a log level (e.g., `ERR`) and take a variable number of arguments.

Stopping execution. Another common strategy for handling failures is to abort the execution of the offending process. In this case, when the system enters an erroneous state that fails a security check, the process takes a safe route to terminate itself. Termination is a safer alternative to avoid harmful side effects such as memory corruption, inconsistent states, and many others. In the kernel, termination is achieved by calling functions such as `panic()` and `BUG()`. Such functions typically have patterns such as dumping the contents of the stack, printing the trace, and self-crashing.

Fixing errors instantly. In the event of failure, a few shallow errors are handled locally in the current function by instantly fixing the erroneous value. After the instant fix, the execution can continue without termination or error handling. Such an instant fix is uncommon and is usually limited to simple cases such as assigning a previously cached value to a potentially changed variable or resetting a variable to a boundary value.

3 CHEQ: Identifying Security Checks

Since checking failures requires handling which has clear patterns as summarized in §2, CHEQ automatically identifies security checks through identifying FH. In this section, we present the approach of CHEQ and the design of its key components.

3.1 A Motivating Example

We first use the example shown in Figure 1 to illustrate how to identify security checks through FH primitives. The goal is to identify which conditional statements (lines 2, 4, and 9) are security checks. Each conditional statement has two branches. The key challenge is to decide whether their branches have FH primitives. Since the code is straightforward, we can quickly analyze the code to identify FH primitives and security checks. Specifically, one branch of the conditional statement at line 9 calls `BUG()`, an error-handling function, thus is marked as MUST-FH. However, its other branch assigns the return value of `put_page()` to `ret`, which may or may not be an error code. Therefore, this branch is marked as MAY-FH. Figure 2 shows the control-flow edges marked with the corresponding FH properties. Based on the definition of security checks, the conditional statement at line 9 is identified as a security check. Similarly, the conditional statement at line 4 is a security check. However, the conditional statement at line 2 is not a security check because neither of the branches is marked as MUST-FH.

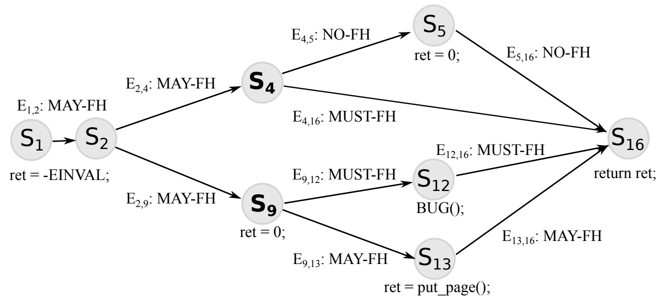

Fig. 2: The corresponding FH graph of the code in Figure 1. S: statement; E: edge connecting two statements. Each edge is marked with an FH property. Conditional statements `S4` and `S9` are identified as security checks based on the definition; however, `S2` is not a security check.

While identifying the security checks in the given example is easy, it becomes challenging when the code is complicated. We identify two main challenges as follows.

- **C1:** Comprehensively identifying FH primitives. OS kernels have many custom error codes and error-handling functions. Automatically identifying them is challenging.
- **C2:** Interactively marking FH properties for error returning. Since error codes are often propagated and changed along code paths across functions. Deciding whether a branch finally ends with returning an error code can be complicated.

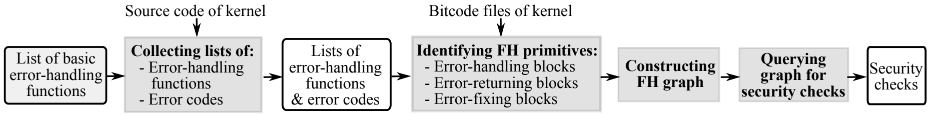

Fig. 3: The workflow of CHEQ. CHEQ takes as inputs the source code of the target kernel and an initial list of basic error-handling functions, and then automatically identifies security checks.

3.2 Approach Overview

Figure 3 shows the workflow of CHEQ. CHEQ takes as inputs the source code of an OS kernel and a list of *basic* error-handling functions of the kernel as a priori knowledge.

Note that error-handling functions include both the ones for stopping the execution and the ones for issuing error messages. The first step of CHEQ is to comprehensively identify error-handling functions and error codes. Based on the provided list of basic error-handling functions, CHEQ employs two automated approaches to augment the lists of error-handling functions and error codes. With the lists, CHEQ moves to the second step to identify FH primitives in basic blocks. Although instant error fixes in the current function are uncommon, CHEQ also identifies them with the patterns described in §2.2. After identifying all basic blocks with FH primitives, CHEQ constructs the FH graph with edges marked with properties MUST-FH, MAY-FH, and NO-FH. At last, CHEQ queries the graph to identify security checks.

3.3 Finding Error-Handling Functions and Error Codes

The first step of CHEQ is to augment the lists of error-handling functions (functions for stopping execution and issuing error messages) and error codes.

```
 /* VM_FAULT_ERROR is a custom error code */
if (unlikely(fault & VM_FAULT_ERROR))
    /* Call error-handling function BUG() */
    BUG();
if (ret < sizeof(*reply)) {
    /* Call error-handling function */
    pr_err("bad size %d \n", ret);
    /* EINVAL is a standard error code */
    return -EINVAL;
}
```

Fig. 4: Two ways of associating error codes to the calls of error-handling functions.

Collecting error-handling functions. We present two heuristic-based approaches for automatically collecting more error-handling functions based on the provided basic ones. These two approaches are complementary to each other. Also, they are general and applicable to other system software. (1) *Wrapper-based approach.* An OS kernel has only a small number of basic error-handling functions because they are critical and are often implemented in assembly. For example, `BUG()`, `panic()`, and `dump_stack()` in Unix-like OS kernels are functions for terminating the offending process. Moreover, basic functions `printk()` and `fprintf()` are used for issuing error messages. Developers often write wrappers of such functions to have module-specific error-handling functions. Therefore, by identifying wrappers of the provided list of basic error-handling functions, we can find general error-handling functions. We will present the identification of wrappers in §5.1. (2) *Pattern-based approach.* The wrapper-based approach has a limitation in identifying error-message functions. Functions `printk()` and `fprintf()` become error-message functions only when their arguments have certain patterns (e.g., containing `KERN_ERR` and `stderr`). To complement the wrapper-based approach, we also use patterns. OS kernels provide severity levels for logging messages. For example, with the Linux convention, error-handling functions typically have a name or an argument with a severity level. Specifically, levels `0-4` (`KERN_EMERG`, `KERN_ALERT`, `KERN_CRIT`, and `KERN_ERR`) indicate critical issues. Further, such functions take a variable number of arguments. Detecting these patterns is straightforward for a static analysis tool. We present the details in §5.1.

Collecting error codes. To identify error returning, we need to decide if the returned value is an error code, which requires us to first know defined error codes. While standard error codes are defined in dedicated header files (e.g., `errno.h`), custom error codes scatter across numerous files over different modules. Manually collecting custom error codes is impractical; we thus propose a general approach to automatically find both standard and custom error codes. We observe that error codes are macro constants and often appear together with calls of error-handling functions. Figure 4 shows two ways of how error codes and error-handling functions are associated. In the first case, if a macro constant (e.g., `VM_FAULT_ERROR`) is checked in a conditional statement, and a branch calls an error-handling function (e.g., `BUG()`), the macro constant is an error code. This is sensible because the macro constant acts as an error indicator. In the second case, if a macro constant is returned in a basic block that calls an error-handling function (e.g., `pr_err()`), the macro constant is also an error code. This is because the basic block is in an error path for handling the error. Using these associations, we can automatically find both standard and custom error codes based on error-handling functions.

3.4 Identifying FH Primitives

After collecting the lists of error-handling functions and error codes, CHEQ moves to the second step to identify FH primitives. In this step, CHEQ analyzes the bitcode files of the target kernel and identifies LLVM basic blocks that have FH primitives.

Identifying error-code returning. Even with the list of error codes, identifying error-returning primitives can be complicated. Not every error code used in a function will be returned, and whether a value is an error code depends on some context. For example, `NULL` in an assignment instruction is an error code only when the assignment targets a return value of the function. More importantly, error codes can be propagated and changed across functions. To precisely identify error-returning primitives, a data-flow analysis is required to decide whether a return instruction finally returns an error code.

CHEQ begins the data-flow analysis by identifying return instructions in each function F of the kernel. To decide whether the return value RV_F of a return instruction is an error code, CHEQ backwardly analyzes values propagating to it. Specifically, CHEQ identifies all sources of RV_F. The sources can be a constant, a local variable, a global variable, or an argument of F. If a source is a constant error code, CHEQ determines that the basic block that contains the instruction (e.g., `store`) assigning the error code to the return value has an error-returning primitive. Because the returned value is decided by the last assignment, CHEQ stops the backward data-flow analysis once a source is found. CHEQ's analysis is inter-procedural. That is, error codes coming from called functions will also be covered.

In our current implementation, we chose *not* to use alias analysis [5] for the data-flow analysis because we observed that error-code propagation is straightforward and typically does not involve complicated point-to relationships. For instance, we did not see any example in our study in which an error code is stored to memory through a pointer and later loaded through a different pointer. If we experience non-negligible false reports due to aliasing, we can include alias analysis in the future, which is orthogonal to CHEQ.

Identifying error handling and fixing. Identifying error-handling primitives that issue error messages or stop the execution is straightforward with identifying the calls of corresponding functions. In this section, we present how to identify instant error-fixing cases that do not return an error code or call an error-handling function. The insight we use to identify instant error fixes is that such fixes are usually limited to simple cases such as resetting a variable with an old value or a boundary value. Based on our observation in §2, in these cases, two variables are compared in a conditional statement, and then one is assigned to the other. Therefore, if a conditional statement compares two variables a and b, and $a = b$ or $b = a$ occurs in the basic block following the conditional statement, we identify it as an instant error fix. Note that such an analysis is by no means complete but can only cover most error-fixing cases we observed in our study.

3.5 Constructing FH Graph

After collecting all basic blocks that have FH primitives, the next step is to analyze the branches of a conditional statement to see if they satisfy the definition of security check. To facilitate the analysis of the FH properties of all branches, we propose to construct FH graphs—augmented control-flow graphs whose edges are marked with FH properties. Figure 2 shows an illustrating example of the FH graph. The graph is constructed using an iterative algorithm. Initially, all edges are marked as NO-FH. The algorithm then traverses the control-flow graph from the function entry. When reaching a basic block with FH primitives, the algorithm updates the properties of forwardly and backwardly reachable edges based on the following policy.

- For error-fixing and error-handling primitives, since they do not "propagate", we update properties of only the incoming and outgoing edges of the current basic block.
- For error-code returning primitives, since error codes propagate, we iteratively update properties of *all* edges reachable to and from the current basic block. Properties are merged upon branches based on Equation 1. Note that, we have a dedicated flag to differentiate error handling and error returning.

3.6 Identifying Security Checks

With the FH graph, identifying security checks is simplified as querying the FH properties of the outgoing edges of conditional statements. A conditional statement that has at least one MUST-FH outgoing edges and at least one MAY-FH or NO-FH outgoing edges is identified as a security check. In the example in Figure 2, by checking the properties of $E_{4,5}$, $E_{4,16}$, $E_{9,12}$, and $E_{9,13}$, we identify conditional statements S_4 and S_9 as security checks. However, S_2 is not a security check because neither of its branches is MUST-FH.

4 Detecting Critical Kernel Semantic Bugs

Security checks are very informative in revealing critical semantics. Without such information, a bug detected by previous approaches [2, 6–8, 12, 25, 31, 35] may not be critical at all or is unlikely to be triggered. In this section, we will present how to use this information to effectively detect critical semantic bugs involving NULL-pointer dereferencing, missing error handling, and double fetching, in the OS kernels.

Categorizing security checks. Before we identify semantic bugs, we first categorize security checks into three classes: NULL check (checking a pointer against `NULL`), error-code check (checking a variable against an error code), and value check (all other cases).

To identify if the condition in a security check is an error code, we will further analyze the source of the checked variable to ensure that it may come from an error code. After categorizing security checks, we use them to detect each class of semantic bugs.

4.1 Detecting Null-Pointer Dereferencing

NULL-pointer dereferencing is a case in which a pointer that is potentially NULL is dereferenced without a NULL check. NULL-pointer dereferencing will typically result in crashes of a program. In OS kernels, NULL-pointer dereferencing will at least result in the termination of the offending process, leading to Denial-of-Service.

Detection approach. We observe that NULL pointers, in most cases, come from the return values of functions (e.g., memory allocators). Our detection for NULL-pointer dereferencing consists of three steps: (1) identifying functions that may return NULL pointers, (2) identifying cases where the returned pointers of the callsites of such functions are directly dereferenced without a NULL check, and (3) ranking the cases based on the likelihood of the functions returning a NULL pointer.

Analyzing NULL pointers. An input of the detection is the NULL checks identified by CHEQ. By backwardly tracking (i.e., backward data-flow analysis) the sources of the checked pointers, we first identify which functions return the NULL pointers. With this analysis, we collect a list of functions that may return NULL pointers. After that, we thoroughly identify all the callsites of these functions and forwardly track their return values (pointers) to decide whether they are NULL-checked and dereferenced. Note that all the analyses are inter-procedural.

Ranking based on statistics. After the analysis, our detection generates two global maps for these functions. One map records how many callsites of a function do not have NULL checks, and the other map records how many callsites of the function do have NULL checks. With these two maps, we generate the “no-check” ratio for each function. To generate the bug report, we rank the callsites that do not have NULL checks in an ascending order, based on the ratio. Cases with a lower no-check ratio are more likely to be real bugs. Functions with a 100% or zero ratio are excluded from the ranked list.

```
/* drivers/target/target_core_rd.c */
static ssize_t rd_set_configfs_dev_params() {
  int arg;
  /* arg is uninitialized if match_int fails*/
  match_int(args, &arg);
  /* ERROR: page refcount can be undefined */
  rd_dev->rd_page_count = arg;
}
int match_int(substring_t *s, int *result) {
  buf = match_strdup(s);
  if (!buf) return -ENOMEM;
  val = simple_strtol(buf, &endp, base);
  if (endp == buf) return -EINVAL;
  *result = (int) val;
}
```

Fig. 5: A new missing error handling bug.

```
/* File: drivers/scsi/sg.c */
/* First data fetch from buf to opcode */
__get_user(opcode, buf);

/* Security check against opcode */
if ((opcode >= 0xc0)
        && old_hdr.twelve_byte)
    cmd_size = 12;

/* Second data fetch from buf to cmnd */
if (__copy_from_user(cmnd, buf, cmd_size))
    return -EFAULT;
/* First byte in cmnd could have been
 * changed in user space */
sg_common_write(..., cmnd, ...);
```

Fig. 6: A new double-fetch bug.

4.2 Detecting Missing Error Handling

We next detect missing error handling. Figure 5 shows an example of such bugs. Line 5 incorrectly assumes the success of `match_int()`, by not checking its return value. However, this function has multiple ways to fail (line 11 and 13) and returns different error codes. When `match_int()` fails, `arg` is not touched and left uninitialized, which may contain random or even malicious value [13]. Using `arg` as a refcount (line 7) will result in memory exhaustion or denial-of-service attacks [14]. In fact, since such bugs completely miss the intended security checks, they may result in many other critical issues such as data losses, information leaks, or even privilege escalation [15, 25, 35].

Detection approach. Similar to detecting NULL-pointer dereferencing, we identify functions that might return error codes (both standard and custom error codes) and use data-flow analysis to determine whether these returned error codes are handled. We mark a returned error code as "handled" when it is both security-checked and handled with at least one of the FH primitives described in §2.2. Similarly, we record how many returned error codes of a function are handled and how many are not handled, and rank the unhandled cases based on the ratio.

4.3 Detecting Double-Fetch Bugs

In addition to checking critical function calls, security checks also reveal the constraints that should be enforced to the uses of a variable. We further exploit such information to detect double-fetch bugs [6, 23, 31, 32, 34]. Figure 6 shows a new double-fetch bug found with CHEQ. The first byte in `buf` is fetched twice and checked in between. If a malicious multi-threaded user-space program races to change the byte, line 15 may use a value smaller than `0xc0`, thus invalidating the security check.

Detection approach. We detect double-fetch bugs in three steps: (1) identifying the sources of checked variables; (2) if the sources are data fetches (e.g., `copy_from_user()`) from the user space, analyzing whether the source data is fetched again, and (3) deciding that the constraints (i.e., security checks) are not enforced before the newly fetched data is used. Compared to existing detection [31, 34], a unique strength of our detection is that it uses CHEQ to target only cases in which the fetched data is critical. That is, invalid values of fetched data may result in critical errors. Such detection eliminates non-critical cases which are very common, as reported by both Deadline [34] and Wang et al. [31].

5 Implementation

We have implemented CHEQ and bug detection as LLVM (of version 8.0.0) passes. We also have an LLVM pass for constructing a global call-graph for all kernel modules. Error-handling functions and error codes are collected through both Python script code and LLVM data-flow analysis, and saved to the configuration file of CHEQ. The report of CHEQ includes details about the identified security checks, including the conditional statement and its source code, the corresponding source line, and the FH primitives and their source code. We now present some interesting implementation details.

5.1 Collecting Error-Handling Functions

We implement two heuristic-based approaches to augment the list of error-handling functions. In the current implementation, the initial list contains only 10 basic error-handling functions such as `BUG()` and `panic()`, and our following two approaches substantially found additional 531 in the Linux kernel and 93 in the FreeBSD kernel.

Finding wrappers. First, we identify functions that internally call the provided basic error-handling functions. Second, we ignore non-functional code such as message-printing code. Third, if a function does not call other functions but just basic error-handling functions, we identify it as a wrapper. The wrapper-based approach is mainly for identifying error-handling functions that stop the current execution.

Finding patterns. We also rely on patterns to find custom error-message functions. We collect all the functions that have a name ending with the error level (`0-4`) and a variable number of parameters. To minimize false positives, we further rely on the call-graph to ensure that general printing functions `printk` and `fprintf` are internally called.

5.2 Preparing LLVM IR for CHEQ

Compiling source code. Compiling the OS kernels like the Linux kernel into LLVM IR often has compatibility issues. Since CHEQ is a code-analysis tool instead of a code-instrumentation tool, we choose to discard modules that cannot be compiled successfully. For the Linux kernel, we compiled 17,343 modules (source-code files) with the `allyesconfig` option; only 7 modules failed. For the FreeBSD kernel, we compiled 1,483 modules, without any failure case, with the `GENERIC` configuration. To preserve the original code patterns in the LLVM IR as much as possible, which is useful for debugging purposes and understanding how functions are called, we compiled the source code using the `-O0` optimization and completely disabled inlining by modifying Clang. We use debug information to differentiate macro constants from general integers in LLVM IR. To find targets of an indirect call, we use the signature-based approach [19, 30] to map address-taken functions to indirect callsites. We unroll loops once. That is, we treat `while` and `for` statements as `if` statements. Such simplification would not affect the accuracy of CHEQ because the FH properties are independent irrespective of the number of iterations.

6 Evaluation

In this section, we evaluate CHEQ and our bug detection by applying them to the Linux kernel of version 4.20.0-rc5 with the top git commit number `b72f711a4efa` and the FreeBSD-13 kernel with the top git commit number `d2e46ebc0d4`.

6.1 Evaluating Security-Check Identification

Statistical results. Before presenting the evaluation results, we first present some interesting statistical numbers in Table 1. The analysis covered 10.2 and 1.2 million lines of code for the Linux kernel and the FreeBSD kernel respectively, reported by the tool `cloc`. For Linux, CHEQ identified 447K security checks from 1.6 million general

Kernel	SLOC	Files	Conditional statements	Security checks	Through error codes	Through EH function	Through error fixing
Linux	10.2M	17K	1.6M	447K	362K	137K	3,350
FreeBSD	1.2M	1.5K	139K	25K	17K	9K	536

Table 1: Some security check–related statistical numbers reported by CHEQ.

conditional statements. 137K (28%) security checks are identified through error-handling functions or error-fixing code. Instant error fixing is uncommon; CHEQ found only 3,350 security checks via error fixes. The combination is larger than the total number because a security check may have multiple FH primitives. For FreeBSD, CHEQ identified 25K security checks from 139K conditional statements, 9K (35%) of which were identified through error-handling functions or error-fixing code. Such security checks cannot be identified through error-code returning.

False negatives. CHEQ identifies security checks via the existences of FH primitives. CHEQ may have false negatives if not all FH primitives are precisely identified. To find false negatives, we used the ground-truth set of security checks collected in §2. The set consists of 600 security checks in total. All of the corresponding conditional statements are present in the Linux kernel of the version in our experiments. We used a script to match the ground-truth security checks with the ones reported by CHEQ, through source-code line numbers. If one cannot be matched, we mark it as a false negative, and then analyze the causes manually.

In total, CHEQ identified 585 (97.5%) security checks and missed 15 (5 are from previous missing-check patches, and 10 are from the set we manually collected). We analyzed the causes of false negatives. The main cause is that the FH primitives used in CHEQ failed to cover some special cases. Specifically, eight cases handle failures by just returning "`void`" or `false` instead of an error code. Five cases just release the system resources (e.g., calling `release_mem_region()`) upon failures and do not return an error or issue an error message. The last two cases store the error code to a global variable or an argument instead of the return value. CHEQ can be improved to eliminate the last two cases by including arguments and globals as error-code propagation channels.

False positives. CHEQ identified 447K and 25K security checks for Linux and FreeBSD kernels. To evaluate false positive, we randomly selected 500 and 100 security checks reported by CHEQ for each kernel, and confirmed whether they are real security checks based on the definition of security checks (§2.1). Note that our definition of security checks does not adhere to any specific program, so the confirmation would not suffer from overfitting problems. The results show that 590 (98.3%) of them are true positives.

We then analyzed the causes of the 10 false positives. Specifically, the most common cause (five cases) is misidentifying custom error codes. Another common cause (three cases) is that some conditions are always true. For example, `WARN_ON(cond)` logs an error when `cond` is evaluated to be true. Therefore, `WARN_ON(1)` is not a security check, but will be a false positive in CHEQ when the target program is compiled with the `O0` optimization level. This is an easy-to-fix problem in CHEQ because we can identify such conditions with compiler optimization. The remaining cases are caused by inaccurate identification of indirect-call targets. As described in §3.4, CHEQ identifies external error codes by inter-procedurally tracking external functions. We may eliminate such false positives by not tracking indirect calls; however, it will introduce false negatives.

Scalability. The experiments were performed on Ubuntu 16.04 LTS with LLVM version 8.0 installed. The machine has a 64GB RAM and an Intel CPU (Xeon R CPU E5-1660 v4, 3.20GHz) with 8 cores. CHEQ is fast—it finished the whole analysis for either Linux or FreeBSD within three minutes: two and half minutes are for loading bitcode files and constructing call-graph, and only half minute is for the identification of security checks.

Generality. CHEQ requires only the LLVM-compilable source code and the initial list of basic error-handling functions (§3.3) in the target program. Since OS kernels typically have clear patterns for error-handling functions, we expect that the manual effort for collecting error-handling functions in new system software is small. To confirm this, we manually investigated Unix-like kernels including OpenBSD, and NetBSD, Darwin-XNU (MacOS), ReactOS (Windows-like), and Chromium browser. We found that all these systems have clear patterns for error-handling functions. Specifically, Unix-like kernels and Darwin-XNU have the same patterns as the Linux kernel, so CHEQ can be directly used to find their error-handling functions. ReactOS and Chromium browser have different but clear patterns of error-handling functions (e.g., `FIXME()` and `LOG(ERROR)`) and thus require only limited manual effort for collecting them. We also found that all these programs have dedicated header files that define standard error codes.

6.2 Evaluating Semantic-Bug Detection

To demonstrate the usefulness of security checks, we describe the results of our bug detection on both the Linux and FreeBSD kernels. Due to limited space, we will primarily focus on the details for the Linux kernel.

NULL-pointer dereferencing bugs. To balance false positives and false negatives, we carefully use a threshold of 10% for the "no-check" ratio to report potential bugs (without the threshold, CHEQ reported in total 3,400 cases). This returns us the top 280 entries in the list. We manually confirmed these entries and found 21 new bugs, as shown in Table 3. False positives are mainly caused by aliasing issues and "can't fail" cases such as using the `__GFP_NOFAIL` flag in allocations, so that they will not fail. We reported all bugs by submitting patches, and Linux maintainers have fixed 7 bugs with our patches.

Missing error handling bugs. To evaluate CHEQ's effectiveness in detecting error-handling bugs, we again set 10% as the no-check ratio, which returns us 682 potential bugs. We manually analyzed them and have confirmed 125 new error-handling bugs. We have submitted the patches for these 125 bugs, and Linux maintainers have accepted 78 (applied 70 and confirmed 8) of them. The main cause of false positives is that the errors are unlikely to occur in the given contexts. In comparison, when we disable CHEQ and generally detect cases where a returned error code is not checked or handled, the detection reports 507,043 cases (the number is only 8,744 when CHEQ is enabled), making bug-confirmation infeasible, which shows the usefulness of CHEQ in eliminating false reports by focusing on erroneous and critical cases only.

Table 4 indicates that drivers code contains about 80% of the bugs. From our interaction with the maintainers, we have some interesting findings. First, the interaction between the kernel and hardware is highly unreliable, thus requiring frequent security checks. Second, if failures occur in an `exiting` stage such as powering down of a device, maintainers believe the security checks for failures are unnecessary. Further, bug fixing resembles the existing error handling protocol. For example, developers inaccurately handle bugs in protocol functions having the `void`-type return values. This is probably because developers could not return an error code in such functions and instead log these with the corresponding driver. These results confirm the viability of CHEQ to find critical and common semantic bugs in OS kernels.

Double-fetch bugs. Although double-fetch bugs have been extensively detected recently, our detector still reports 66 potential double-fetch cases in the Linux kernel; 12 of them have been confirmed as real bugs, and three has been fixed. We did not find any double-fetch cases in the FreeBSD kernel. The details of the new bugs are shown in Table 2. Most false positives are caused by inaccuracy in identifying overlapping in the two fetches. Such false positives can be eliminated with symbolic execution, as shown in [34]. The unique strength of our detection is that it employs CHEQ to find the checked "first" fetches, so the checked variables must be critical. This way, it significantly narrows down the analysis scope and simplifies the detection. We further reverted the patches for 40 double-fetch bugs reported in [31, 32, 34]; some of them are still not patched yet. Our detection successfully reported all these bugs, confirming the detection effectiveness.

We believe that the promising bug-detection results mainly benefit from CHEQ's identification of security checks, which helps automatically infer what are critical and erroneous, and thus eliminate overwhelming false reports.

7 Related Work

To the best of our knowledge, CHEQ is the first to identify security checks. We identify two research lines that are related to CHEQ: analyzing error-code propagation and handling, and detecting missing-check bugs.

Error-code propagation and handling. A handful of research works have investigated bugs in error-code propagation and handling. EIO [4] and Rubio-González et al.[21] proposed static analysis techniques to detect incorrect error code propagation in Linux file and storage systems. EPEx [7] and APEx [8] identify code paths in a callee function that may return error codes and check if the error codes are handled in callers. These tools conservatively identify a return value as an error code as long as it falls in the specified range of error codes. For example, they consider any value $\leqslant 0$ an error code in `OpenSSL`. ErrDoc [29] relies on EPEx [7] to find potential error-handling bugs, and then further diagnose and fix them. Hector [22] finds error-returning paths based on standard or user-specified error codes, and detects resource-release bugs along with the paths.

CHEQ has a different research goal from these tools—identifying security checks that can be used for detecting various classes of semantic bugs. Detecting error-handling bugs is just one application of CHEQ. Even for the part of detecting error-handling bugs, CHEQ differentiates itself from these tools. First, these tools check only error-code returning and propagation, but not calls to error-handling functions or error-fixing code which are covered in CHEQ. Our evaluation (§6.1) shows that about 30% of security checks cannot be identified through error-code returning. Second, they do not have an accurate way of identifying custom error codes, which is an important challenge overcome in CHEQ. Last but not least, CHEQ can specifically identify which conditional statement results in the error-code returning, which can be challenging when error-codes are propagated and changed along the code paths across functions.

Missing-check detection. Vanguard [25] detects missing checks for only four specified critical operations such as arithmetical division. Our bug detection is agnostic about the operations and can detect missing checks for all functions across the kernel. LR-San [32] detects lacking-recheck bugs. It however uses only standard error codes to find checks without considering custom error codes or other FH primitives; therefore, it identified only 131K security checks in the Linux kernel. In comparison, CHEQ is comprehensive and precise in identifying various classes of FH primitives. Juxta [15] utilizes cross-checking to detect semantic bugs such as missing checks in the Linux file systems. It requires multiple implementations of the same standard, such as POSIX for file operations; therefore, it cannot detect security checks in modules containing unique implementations. CHEQ helps address the limitation by automatically identifying security checks.

There are also a few complementary detection tools to our bug detection. Specifically, Chucky [35] combines static analysis and machine learning techniques to uncover missing checks by considering the context of the code. AutoISES [28] generates custom security specifications relying on the similarity of data structures for a given security check across C libraries. Rolecast [26] finds new missing security bugs in PHP scripts by detecting rule violations. MACE [17] finds missing authorization checks in web applications by checking authorization state consistency.

8 Conclusion

We presented our study on security checks and CHEQ, an automated tool for precisely identifying security checks in OS kernels. We also presented that, with identified security checks, we can effectively detect semantic bugs. CHEQ identifies security checks by detecting FH primitives using multiple new techniques such as custom error-code identification and FH graph. Evaluation results show that CHEQ has very low false-positive and false-negative rates. CHEQ also offers opportunities for improving the security of OS kernels by significantly narrowing down the analysis scope to enable expensive and precise analysis techniques. With CHEQ, we detected three classes of critical and common semantic bugs: NULL-pointer dereferencing, missing error handling, and double fetching. We have found 164 new bugs in the Linux and FreeBSD kernels, most of which have been fixed with our patches by maintainers. We believe that the identification of security checks could facilitate future research on semantic-bug detection.

9 Acknowledgment

We would like to thank the anonymous reviewers for their helpful suggestions and comments. This research was supported in part by the NSF award CNS-1815621. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

Bibliography

[1] Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., Adve, V.: Nested kernel: An operating system architecture for intra-kernel privilege separation. In: Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Istanbul, Turkey (Mar 2015)

[2] Dillig, I., Dillig, T., Aiken, A.: Static error detection using semantic inconsistency inference. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '07 (2007)

[3] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z.: Collafl: Path sensitive fuzzing. In: Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland). San Francisco, CA (May 2018)

[4] Gunawi, H.S., Rubio-González, C., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Liblit, B.: Eio: Error handling is occasionally correct. In: FAST. vol. 8, pp. 1–16 (2008)

[5] Hardekopf, B., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). San Diego, CA (Jun 2007)

[6] Institute, I.: Exploiting Windows Drivers: Double-fetch Race Condition Vulnerability (2016), http://resources.infosecinstitute.com/exploiting-windows-drivers-double-fetch-race-condition-vulnerability

[7] Jana, S., Kang, Y.J., Roth, S., Ray, B.: Automatically detecting error handling bugs using error specifications. In: USENIX Security Symposium. pp. 345–362 (2016)

[8] Kang, Y., Ray, B., Jana, S.: Apex: Automated inference of error specifications for c apis. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 472–482. ACM (2016)

[9] Kim, S.Y., Lee, S., Yun, I., Xu, W., Lee, B., Yun, Y., Kim, T.: Cab-fuzz: Practical concolic testing techniques for COTS operating systems. In: Proceedings of the 2017 USENIX Annual Technical Conference (ATC). Santa Clara, CA (Jul 2017)

[10] Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS). Toronto, ON, Canada (Oct 2018)

[11] Koning, K., Chen, X., Bos, H., Giuffrida, C., Athanasopoulos, E.: No Need to Hide: Protecting Safe Regions on Commodity Hardware. In: Proceedings of the 12th European Conference on Computer Systems (EuroSys). Belgrade, Serbia (Apr 2017)

[12] Kremenek, T., Twohey, P., Back, G., Ng, A., Engler, D.: From uncertainty to belief: Inferring the specification within. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. OSDI '06 (2006)

[13] Lu, K., Walter, M., Pfaff, D., Nümberger, S., Lee, W., Backes, M.: Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In: Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA (Feb–Mar 2017)

[14] Mao, J., Chen, Y., Xiao, Q., Shi, Y.: Rid: Finding reference count bugs with inconsistent path pair checking. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. Atlanta, GA (Apr 2016)

[15] Min, C., Kashyap, S., Lee, B., Song, C., Kim, T.: Cross-checking semantic correctness: The case of finding file system bugs. In: Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP). Monterey, CA (Oct 2015)

[16] Mogosanu, L., Rane, A., Dautenhahn, N.: Microstache: A lightweight execution context for in-process safe region isolation. In: Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID). Crete, Greece (Sep 2018)

[17] Monshizadeh, M., Naldurg, P., Venkatakrishnan, V.: Mace: Detecting privilege escalation vulnerabilities in web applications. In: Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS). Scottsdale, Arizona (Nov 2014)

[18] Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Softbound: Highly compatible and complete spatial memory safety for c. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Dublin, Ireland (Jun 2009)

[19] Niu, B., Tan, G.: Modular control-flow integrity. In: Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Edinburgh, UK (Jun 2014)

[20] Peng, H., Shoshitaishvili, Y., Payer, M.: T-fuzz: Fuzzing by program transformation. In: Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland). San Francisco, CA (May 2018)

[21] Rubio-González, C., Gunawi, H.S., Liblit, B., Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C.: Error propagation analysis for file systems. In: ACM Sigplan Notices. vol. 44, pp. 270–280. ACM (2009)

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

[23] Schwarz, M., Gruss, D., Lipp, M., Maurice, C., Schuster, T., Fogh, A., Mangard, S.: Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In: Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS). Incheon, Korean (Jun 2018)

[24] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: Proceedings of the 2012 USENIX Annual Technical Conference (ATC). Boston, MA (Jun 2012)

[25] Situ, L., Wang, L., Liu, Y., Mao, B., Li, X.: Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In: Proceedings of the Tenth Asia-Pacific Symposium on Internetware. p. 5. ACM (2018)

[26] Son, S., McKinley, K.S., Shmatikov, V.: Rolecast: finding missing security checks when you do not know what checks are. In: ACM SIGPLAN Notices. vol. 46, pp. 1069–1084. ACM (2011)

[27] Song, C., Lee, B., Lu, K., Harris, W.R., Kim, T., Lee, W.: Enforcing Kernel Security Invariants with Data Flow Integrity. In: Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA (Feb 2016)

[28] Tan, L., Zhang, X., Ma, X., Xiong, W., Zhou, Y.: Autoises: Automatically inferring security specification and detecting violations. In: USENIX Security Symposium. pp. 379–394 (2008)

[29] Tian, Y., Ray, B.: Automatically diagnosing and repairing error handling bugs in c. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 752–762. ACM (2017)

[30] v. d. Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A tough call: Mitigating advanced code-reuse attacks at the binary level. In: Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland). San Jose, CA (May 2016)

[31] Wang, P., Krinke, J., Lu, K., Li, G., Dodier-Lazaro, S.: How Double-Fetch Situations Turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In:

Proceedings of the 26th USENIX Security Symposium (Security). Vancouver, BC, Canada (Aug 2017)

[32] Wang, W., Lu, K., Yew, P.: Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels. In: Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS). Toronto, ON, Canada (Oct 2018)

[33] Wikibooks: C Programming/Program flow control (2017), https://en.wikibooks.org/wiki/C_Programming/Program_flow_control

[34] Xu, M., Qian, C., Lu, K., Backes, M., Kim, T.: 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 (May 2018)

[35] Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: Exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 499–510. ACM (2013)

[36] Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. In: Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland). Oakland, CA (May 2009)

A Appendix

Subsystem	File	Function	Fetched and checked variable	#Bugs	Status
x86	wmi.c	wmi_ioctl	buf->length	1	S
stm	core.c	stm_char_policy_set_ioctl	size	1	S
scsi	sg.c	sg_write	opcode	1	A
	sg.c	sg_read	old_hdr	1	A
	megaraid.c	mega_m_to_n	signature	1	S
	commctrl.c	aac_send_raw_srb	fibsize	1	S
	dpt_i2o.c	adpt_i2o_passthru	size	2	S
acpi	custom_method.c	cm_write	max_size	1	S
coda	psdev.c	coda_psdev_write	hdr.opcode	2	C
sched	core.c	sched_copy_attr	size	1	A

Table 2: New double-fetch bugs detected with CHEQ. S: Submitted, A: Applied, C: Confirmed.

Filename	Called function	S	K
ci_hdrc_msm.c	of_get_next_available_child	A	L
coh901318_lli.c	dma_pool_create	S	L
fsl-edma-common.c	dma_pool_create	S	L
virtgpu_kms.c	idr_alloc	S	L
virtgpu_vq.c	idr_alloc	S	L
qedr_iw_cm.c	idr_find	A	L
message.c	api_parse	A	L
cx231xx-input.c	i2c_new_device	S	L
cxgb3_offload.c	alloc_skb	S	L
mvpp2_main.c	acpi_match_device	A	L
lag_conf.c	kmalloc_array	S	L

Filename	Called function	S	K
rx.c	kcalloc	S(2)	L
pcie-designware-host.c	alloc_page	S	L
qcom-ngd-ctrl.c	platform_device_alloc	S	L
fw.c	netdev_alloc_skb	S	L
mmal-vchiq.c	vmalloc	S	L
nf_tables_api.c	nla_nest_start	A	L
conntrack.c	nla_nest_start	S	L
rpc_rdma.c	xdr_inline_decode	C	L
netlink_compat.c	genlmsg_put	A	L
iir.c	cam_sim_alloc	S	F
crypto.c	crypto_checkdriver	S	F
ocs_mgmt.c	ocs_malloc	S	F

Table 3: List of new NULL-pointer dereferencing bugs detected with CHEQ. In column S, S, C, and A are Submitted, Confirmed, and Applied patches, respectively. In column K, L , F are the Linux and FreeBSD kernels.

Filename	Called function	S	K
sfi.c	intel_scu_devices_create	S	L
sfi.c	sfi_handle_ipc_dev	S	L
clock_ops.c	pm_clk_acquire	A	L
hci_bcm.c	bcm_init	S	L
hci_intel.c	intel_init	S	L
core.c	devm_hwrng_unregister	A	L
clk-versaclock5.c	vc5_pll_recalc_rate	S	L
sh_cmt.c	sh_cmt_clock_event_resume	S	L
mv_xor.c	mv_xor_channel_add	A	L
hidma_mgmt.c	hidma_mgmt_init	A	L
stm32-mdma.c	stm32_mdma_probe	A	L
memconsole-coreboot.c	memconsole_coreboot_read	S	L
amdgpu_ucode.c	amdgpu_ucode_create_bo	S	L
analogix-anx78xx.c	anx78xx_poweron	S(2)	L
hid-lenovo.c	lenovo_probe_tpkbd	A(2)	L
lm80.c	lm80_probe	A(2)	L
lm80.c	set_fan_div	A	L
xilinx-xadc-core.c	xadc_probe	A	L
ad9523.c	ad9523_setup	A	L
addr.c	ib_nl_ip_send_msg	C	L
sa_query.c	ib_nl_set_path_rec_attrs	S(6)	L
qplib_sp.c	bnxt_qplib_map_tc2cos	A	L
samsung-keypad.c	samsung_keypad_probe	S	L
ad7879.c	ad7879_irq	A	L
elants_i2c.c	elants_i2c_calibrate	C(2)	L
leds-lp5523.c	lp5523_init_program_engine	A	L
drxj.c	drxj_dap_atomic_read_write_block	S	L
drxd_hard.c	InitCC	A(5)	L
drxd_hard.c	SC_ProcStartCommand	A(3)	L
drxd_hard.c	SC_SendCommand	A	L
drxk_hard.c	drxk_get_stats	S	L
lgdt3306a.c	lgdt3306a_read_signal_strength	A(2)	L
mt312.c	mt312_set_frontend	S	L
si2165.c	si2165_wait_init_done	A	L
sp8870.c	sp8870_set_frontend_parameters	A	L
mxl111sf-phy.c	mxl111sf_config_mpeg_in	S	L
cpia1.c	do_command	S(2)	L
cpia1.c	sd_config	S	L
m5602_mt9m111.c	mt9m111_probe	A	L
m5602_po1030.c	po1030_probe	A	L
mc13xxx-core.c	mc13xxx_adc_do_conversion	A	L
sm501.c	sm501_base_init	S	L
atmel-ssc.c	ssc_request	S	L
ics932s401.c	ics932s401_update_device	S	L
bcm_sf2.c	bcm_sf2_sw_mdio_write	A	L
atl1e_main.c	atl1e_mdio_write	A	L
cudbg_lib.c	cudbg_collect_hw_sched	A	L
80003es2lan.c	e1000_init_hw_80003es2lan	A(2)	L
80003es2lan.c	e1000_reset_hw_80003es2lan	A	L
netxen_nic_init.c	netxen_validate_firmware	A	L
mcdi.c	efx_mcdi_set_id_led	A	L
dwmac-sunxi.c	sun7i_gmac_init	A	L
niu.c	niu_pci_vpd_scan_props	A(2)	L
cpts.c	cpts_create	A	L
phy.c	phy_mii_ioctl	S	L
xilinx_gmii2rgmii.c	xgmiitorgmii_read_status	C	L
slic_ds26522.c	slic_read	C	L
wmi.c	ath6kl_wmi_set_roam_lrssi_cmd	A	L
usb.c	brcmf_usb_register	A	L
mesh.c	lbs_persist_config_init	A(2)	L
pci-exynos.c	exynos_pcie_assert_reset	S	L
alienware-wmi.c	alienware_zone_init	S	L
twl4030_charger.c	twl4030_bci_get_property	A	L
palmas-regulator.c	palmas_set_mode_smps	A	L
tps65910-regulator.c	tps65910_probe	A	L
rtc-coh901331.c	coh901331_resume	A	L
rtc-hym8563.c	hym8563_rtc_read_time	A	L
rtc-rv8803.c	rv8803_handle_irq	S(2)	L
dpcsup.c	aac_aif_callback	S	L
qcom-ngd-ctrl.c	qcom_slim_ngd_ctrl_probe	S	L
qcom-ngd-ctrl.c	of_qcom_slim_ngd_register	S	L
hal_init.c	rtl871x_load_fw_cb	S	L
ioctl_linux.c	rtw_wps_start	A	L
ms.c	mspro_rw_multi_sector	C	L
ms.c	ms_copy_page	A(2)	L
ms.c	mspro_stop_seq_mode	S	L
sd.c	reset_sd	S	L
sd.c	sd_execute_write_data	A	L
target_core_rd.c	rd_set_configfs_dev_params	A(2)	L
max310x.c	max310x_uart_init	S	L
adp8870_bl.c	adp8870_bl_ambient_light_zone_store	S	L
sysfs.c	btrfs_sysfs_feature_update	A	L
root.c	proc_root_init	S	L
xfs_super.c	xfs_fs_dirty_inode	S	L
verifier.c	check_func_call	C	L
verifier.c	check_helper_call	C	L
verifier.c	check_ld_abs	C	L
alarmtimer.c	alarmtimer_suspend	S	L
compaction.c	sysctl_extfrag_handler	A	L
hmm.c	hmm_devmem_pages_remove	A	L
main.c	batadv_init	S	L
net_namespace.c	net_ns_init	A	L
route.c	ipv6_sysctl_rtcache_flush	A	L
ip_set_core.c	call_ad	A	L
netlink_compat.c	tipc_nl_compat_sk_dump	A	L
gus_main.c	snd_gus_init_control	S	L
sb16_main.c	snd_sb16dsp_pcm	A	L
ews.c	snd_ice1712_6fire_read_pca	A	L
rt5663.c	rt5663_parse_dp	S	L
sst-mfld-platform-pcm.c	sst_media_hw_params	A	L
pod.c	pod_startup4	A	L
variax.c	variax_startup6	S	L
netback.c	ether_ifattach	S	F
kern_umtx.c	umtxq_check_susp	S	F
scsi_enc.c	cam_periph_runccb	S	F

Table 4: List of new missing error handling bugs detected with CheQ. A number in column S indicates multiple bugs in the module, and S: Submitted, A: Applied, C: Confirmed. In column K, L and F indicate Linux and FreeBSD, respectively.