Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences

Kangjie Lu, Aditya Pakki, and Qiushi Wu, ***University of Minnesota***

https://www.usenix.org/conference/usenixsecurity19/presentation/lu

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences

Kangjie Lu, Aditya Pakki, and Qiushi Wu
University of Minnesota

Abstract

Missing a security check is a class of semantic bugs in software programs where erroneous execution states are not validated. Missing-check bugs are particularly common in OS kernels because they frequently interact with external untrusted user space and hardware, and carry out error-prone computation. Missing-check bugs may cause a variety of critical security consequences, including permission bypasses, out-of-bound accesses, and system crashes. While missing-check bugs are common and critical, only a few research works have attempted to detect them, which is arguably because of the inherent challenges in the detection—whether a variable requires a security check depends on its semantics, contexts and developer logic, and understanding them is a hard problem.

In this paper, we present CRIX, a system for detecting missing-check bugs in OS kernels. CRIX can scalably and precisely evaluate whether any security checks are missing for critical variables, using an inter-procedural, semantic- and context-aware analysis. In particular, CRIX's modeling and cross-checking of the semantics of conditional statements in the peer slices of critical variables infer their *criticalness*, which allows CRIX to effectively detect missing-check bugs. Evaluation results show that CRIX finds missing-check bugs with reasonably low false-report rates. Using CRIX, we have found 278 new missing-check bugs in the Linux kernel that can cause security issues. We submitted patches for all these bugs; Linux maintainers have accepted 151 of them. The promising results show that missing-check bugs are a common occurrence, and CRIX is effective and scalable in detecting missing-check bugs in OS kernels.

1 Introduction

Security checks are a class of conditional statements that validate program execution states. Security checks play an important role in ensuring the security of OS kernels. Not only do OS kernels accept arbitrary untrusted inputs, but they also perform complicated tasks such as concurrent resource management and multi-user/capability access control. Therefore, OS kernels often enter into erroneous states and require security checks to capture them.

A missing-check bug exists when an intended security check is not enforced for a critical variable. Examples of such critical variables include the ones used to indicate potential erroneous execution states, e.g., the return value of `kmalloc()`, and the ones used in critical operations, e.g., the size variable in `memcpy()`. Figure 1 shows a concrete example of missing-check bugs. `ib_get_client_data()` may fail and return `NULL`. Since `smcibdev` is not checked, the following uses of it may cause multiple problems—NULL-pointer dereferences, failures in removing and unregistering devices, and memory leaks. To fix the problem, a security check should be enforced between lines 6 and 8 to ensure that `smcibdev` is not NULL.

```
1  /* Linux: net/smc/smc_ib.c */
2  static void smc_ib_remove_dev(struct ib_device *ibdev...)
3  {
4    struct smc_ib_device *smcibdev;
5      /* ib_get_client_data may fail and return NULL */
6    smcibdev = ib_get_client_data(ibdev, &smc_ib_client);
7      // ERROR1: NULL-pointer deference
8    list_del_init(&smcibdev->list);
9      /* ERROR2: device cannot be removed or unregistered */
10   smc_pnet_remove_by_ibdev(smcibdev);
11   ib_unregister_event_handler(&smcibdev->event_handler);
12     /* ERROR3: memory leak */
13   kfree(smcibdev);
14     /* No return value: caller cannot know the errors */
15 }
```

Figure 1: Example: A new missing-check bug found by CRIX. The missed check against variable `smcibdev` will cause multiple problems, as annotated in the code.

Missing-check bugs may cause critical security impacts because security checks are a main means for OS kernels to ensure their security and reliability. To understand the importance of security checks, we first studied recently reported security vulnerabilities in the National Vulnerabilities Database (NVD). We found that 59.5% security vulnerabilities stem

from missing-check bugs, which were all fixed by inserting security checks. We then investigated these vulnerabilities and found that at least 52% (excluding denial-of-service cases) of them will cause severe security impacts such as permission bypass, memory corruption, system crashes/hangs.

Although missing-check bugs are critical and prevalent, only a few research works have attempted to detect them in OS kernels and have several limitations. In particular, Vanguard [32] assumes that some critical operations should always be checked. It however detects missing-check bugs for only four specified critical operations such as arithmetical division and array indexing. Some other approaches (e.g., Chucky [46], Juxta [23], Kremenek et al. [19], and Dillig et al. [7]) employ cross-checking, inconsistency analysis, or machine learning to reduce false positives in detecting bugs. These approaches have non-trivial limitations. First, the manual specification for critical variables covers only a small set of critical variables. This leads to significant false negatives. Second, most of these approaches are not semantic- or context-aware. For example, they tend to treat any conditional statement (i.e., an `if` or a `switch` statement) as a security check. In fact, whether a variable requires a security check highly depends on its semantics and contexts, without considering which, the detection would suffer from high false-negative and false-positive rates.

The lack of effective research in detecting missing-check bugs is arguably because of several inherent challenges. (1) Critical variables that require security checks take diverse forms. For example, a critical variable can be a parameter of a critical function (e.g., the size variable in `memcpy()`), a global variable, or a return value of a function call that is used in only security checks but not others such as arithmetic operations (this case is missed by Vanguard [32]). Therefore, generally checking for different kinds of critical variables is hard.

(2) Identifying security checks requires semantic understanding. Treating any conditional statement as a security check will cause both significant false positives and false negatives. In fact, according to our study §6, the majority (about 70%) of conditional statements are not security checks but some normal selectors in which both branches of the conditional statements lead to normal execution. (3) Missing-check bugs are context dependent, and the detection should be context aware. For example, an error code may not require a security check at all if it is used in a debugging function. As such, missing-check detection should be context aware. (4) Last but not least, OS kernels are extremely large and complex. Checking every variable will not scale, and corner cases such as hand-written assembly will make the analysis error-prone.

In this paper, we present CRIX (Criticalness and constraints Inferences for detecting missing checks), a system that overcomes the aforementioned challenges to effectively detect missing-check bugs in OS kernels. At a high level, CRIX first employs an automated approach to identify critical variables as the analysis targets. For each critical variable, CRIX constructs *peer* slices that share similar semantics and contexts. After that, CRIX models constraints of conditional statements in each slice. By cross-checking the modeled constraints of the peer slices of a critical variable, CRIX finally identifies deviations as potential missing-check bugs and reports them for further confirmation.

While the high-level idea of CRIX is intuitive, it entails overcoming multiple technical challenges. We thus have developed multiple new techniques to tackles these challenges. (1) We first propose a two-layer type analysis to identify indirect-call targets, which serves as a foundation of our data-flow analysis engine. In addition to the function-type analysis (the first layer) employed by traditional control-flow integrity (CFI) techniques [4, 25, 40], the two-layer type analysis further uses struct-type analysis, which is also employed by Ge et al. [10], to refine indirect-call targets. (2) We then develop an automated analysis that identifies security-checked variables as potential critical variables, which not only narrows down the analysis scope and thus scales the detection to OS kernels, but also significantly reduces false reports by filtering out non-critical variables. (3) We further propose peer-slice construction to collect slices of a critical variable that share similar semantics and contexts. The set of peers enables effective cross-checking for potential missing check cases of the critical variable. (4) At last, to precisely detect missing-check cases, we construct constraints from the conditional statements in the peer slices and model them based on their semantics (e.g., the condition type in conditional statements). The modeled constraints allow CRIX to cross-check slices for detecting missing-check bugs, in a semantic-aware manner.

With the new techniques, CRIX's analysis is scalable, semantic- and context-aware, and the data-flow analysis engine in CRIX is inter-procedural, flow-, context-, and field-sensitive. By focusing on the small set of automatically identified critical variables, CRIX can scale to large programs like the Linux kernel. The peer-slice construction allows CRIX to reason about potential missing-check cases in a semantic- and context-aware manner, and the constraint modeling and cross-checking enable CRIX to infer the criticalness of critical variables. As a result, CRIX is able to effectively and precisely detect missing-check bugs in complex and large system software such as the Linux kernel.

We have implemented CRIX on top of LLVM as multiple static-analysis passes. We chose the Linux kernel as the experimental target given its prevalence and complexity (more than 25 million SLOC). CRIX finished the analysis for the whole Linux kernel in about one hour and reported many missing-check cases. By manually investigating the top 804 missing-check cases reported by CRIX, we confirmed 278 new missing-check bugs. We also submitted patches for all of them to the Linux maintainers. Out of these patches, 151 have been accepted, with 134 applied to the mainline Linux kernel and 17 confirmed. The results show that CRIX is highly scalable and effective in finding missing-check bugs. We also

discuss CRIX's portability in §6 and believe that CRIX can be easily extended to detect missing-check bugs in other system software.

We make the following contributions in this paper.

- **A new system for missing-check bug detection.** Missing-check bugs constitute the root cause of the majority (59.5%) of recent security vulnerabilities. We propose a semantic- and context-aware approach to scalably and effectively detect missing-check bugs in OS kernels. The resulting system, CRIX, is open sourced [1] .

- **Multiple new general techniques.** We propose multiple new general techniques in CRIX, which would benefit other research. In particular, the peer-slice construction identifies code paths that share similar semantics and contexts, which is useful for general differential analysis. The automated critical-variable inference finds a small set of targets that deserve precise analysis and protection, which narrows down target scope and could improve the performance for techniques such as fuzzing and data-flow integrity. The two-layer type analysis refines indirect-call targets, without introducing false negatives, which is also useful for inter-procedural static analysis, control-flow integrity, and program debloating.

- **Numerous new bugs in the Linux kernel.** With CRIX, we found a large number of new missing-check bugs in the Linux kernel, which may cause critical security and reliability issues to the Linux kernel used by billions of devices. We reported these new bugs and have worked with Linux maintainers to fix many of them.

The rest of this paper is organized as follows. We present the study on missing check bugs in §2, the design of CRIX in §4, implementation of CRIX in §5, evaluation of CRIX in §6. We further discuss the extension and limitations of CRIX in §7. We present related work in §8, and conclude in §9.

2 Missing Checks in OS Kernels

To propose an effective approach to finding missing-check bugs, we first study the characteristics of previously reported missing-check bugs.

Bug-set collection. We collect previously reported missing-check bugs from NVD [26]. We first selected the recent 200 vulnerabilities that were reported during 2017 and 2018. Out of them, we then selected the ones fixed by enforcing security checks, which returned us 119 (59.5%) vulnerabilities. We finally took the missing-check bugs leading to these vulnerabilities as the bug set for our study.

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

2.1 Impact of Missing-Check Bugs

To assess the impact of missing-check bugs, we investigated (1) what percent of security vulnerabilities are caused by missing-check bugs, (2) common classes of security vulnerabilities caused by missing-check bugs, and (3) severe security impact of missing check–related vulnerabilities.

Percent of missing check–related vulnerabilities. As we mentioned in the bug-set collection, a majority (59.5%) of recent security vulnerabilities were caused by missing-check bugs. This is expected because common vulnerabilities such as out-of-bound access and access-control errors are typically fixed with security checks.

Common classes of missing-check impact. We then classified the security impact of the 119 missing-check bugs based on the classification provided by CVEDetails [48]. We found that missing-check bugs can introduce at least ten classes of vulnerabilities. Table 1 shows the six most common classes. In particular, more than half of the missing-check bugs may result in denial-of-services, and more the 52% of them may result in other severe impacts. Some missing-check bugs may have multiple impacts, so the total number is $> 100\%$.

DoS	Over flow	Bypass privi.	Info leak	Memory corrupt	Code exec.
51.2%	16.0%	14.3%	11.7%	6.7%	3.4%

Table 1: Common security impacts of missing-check bugs.

Severe security impact. We also looked into the most severe vulnerabilities from 2017 to 2018 that have a CVSS (Common Vulnerability Scoring System) score 10 (the highest severity level) from the Linux kernel. We in total found 15 such vulnerabilities. Specifically, we found that 11 of these vulnerabilities are caused by missing-check bugs. The targets of the missing checks in these vulnerabilities include buffer length, function return value, pointer value, and permissions. Correspondingly, the missing-check bugs will cause severe impacts, including buffer overflow, use-after-free, memory corruption, permission pass, which will finally result in data losses, information leaks, and even attackers control of the whole system. Figure 2 shows an example of a severe missing-check bug (CVE-2017-18017) with a CVSS 10. The attacker-controllable `len` and `tcp_hdrlen` are used as a loop-termination condition for memory access. Missing the security checks for these variables will result in denial of service, information leak, and memory corruption.

2.2 Targets of Security Checks

According to our analysis of the missing-check bugs, generally, security checks have two classes of targets: state variables and critical-use variables.

```
/* Linux: net/netfilter/xt_TCPMSS.c (CVE-2017-18017) */
static int tcpmss_mangle_packet(struct sk_buff *skb,
        unsigned int tcphoff, ...) {
    tcph = (struct tcphdr *)(skb_network_header(skb) + tcphoff);
    tcp_hdrlen = tcph->doff * 4;
    /* Security checks for both "len" and "tcp_hdrlen" */
    if (len < tcp_hdrlen || tcp_hdrlen < sizeof(struct tcphdr))
        return -1;
}
```

Figure 2: A missing-check bug causing multiple severe security impacts: denial of service, information leak, and memory corruption. The bug is assigned with ID CVE-2017-18017.

State variables. State variables indicate if the current execution is in an erroneous state, e.g., if an operation is successful or not. According to the C programming convention, a return value of a function often serves as an indicator of execution states. State variables are prevalent in OS kernels because kernel operations are error-prone. OS kernels have to frequently use and check state variables to ensure that an operation is successful. A special feature of state variables is that they are often used in only security checks but not any other function-related operations such as arithmetic operations. Line 4 in Figure 3 is an example of checking the state variable `ret`, which is used only in the security check at line 4. In function `btrfs_search_slot()`, different error codes are returned for indicating various erroneous states, which should be checked in callers.

Critical-use variables. Variables used in critical operations are another common class of check targets. Intuitively, variables should be checked before being used in a critical operation. Common critical-use variables include pointers in dereferencing, offsets in array indexing, operands of binary operations such as arithmetic division, and parameters in critical functions (e.g., `memcpy()`) that may cause security issues. Line 9 in Figure 3 is an example of checking the variable `tx_out` before it is being used in the critical function `dmaengine_submit()` which internally dereferences `tx_out`.

```
/* Linux: fs/btrfs/inode-map.c */
ret = btrfs_search_slot(NULL, root, &key, path,0,0);
/* "ret" is a state variable for the search operation */
if(ret < 0)
    goto out;

/* Linux: drivers/crypto/omap-des.c */
/* "tx_out" is checked before "dmaengine_submit" uses it */
if(!tx_out)
    return-EINVAL;
dmaengine_submit(tx_out);
```

Figure 3: Examples of check targets. `ret` is a state variable checked in line 4, and `tx_out` is checked at line 9 because it is used by the critical function `dmaengine_submit()`.

3 Overview of CRIX

The goal of CRIX is to detect missing-check bugs in OS kernels. To this end, CRIX automatically infers whether a variable in the target OS kernel requires a security check. The detection of missing-check bugs, in general, is to answer the following questions: (1) does a variable require a security check, (2) if possible, what security check should be enforced for the variable, and (3) is such a security check present. Answering these questions requires the understanding of the contexts and semantics of the code, which is challenging. Prior research [41] has shown the promise of statistical inferences in finding bugs. Such inferences identify inconsistent cases as potential bugs, which avoids the hard problem of understanding contexts and semantics. It makes sense that a deviation from common patterns is often problematic and thus is likely a potential bug, given that a majority of the code is correct. In CRIX, we also employ the general idea of statistical inference to find missing-check bugs. However, compared to the previous detection, CRIX is context and semantic aware.

Figure 4 shows the overview of CRIX. CRIX consists of three phases: (1) preprocessing phase which prepares a global call graph, control-flow graph, and alias results; (2) analysis phase which performs the key analyses to identify critical variables, construct peer slices for them, and construct constraints for peer slices, and (3) postprocessing phase which cross-checks constraints of peer slices and reports missing-check bugs.

In the first phase, given the LLVM IR (intermediate representation), CRIX constructs a precise global call graph, which is not only foundational to all the following data-flow analysis, but also enables the peer-slice construction, as will be presented in §4.3. Since LLVM does not provide targets of indirect calls, CRIX employs a technique, namely two-layer type analysis, to precisely find indirect-call targets.

In the second phase, CRIX first identifies critical variables. Because a large number of variables are non-critical in OS kernels, conservatively checking all variables would cause significant scalability and false-positive issues. CRIX therefore first identifies critical variables (see §2.2). The intuition behind the critical-variable identification is that security-checked variables are typically critical. Therefore, CRIX identifies the security-checked variables as critical variables, which however requires CRIX to first identify security checks. Since the majority of conditional statements are not security checks [43], CRIX employs an approach to identify security checks, as presented in §4.2.1.

Since the critical variables identified through security checks have already been checked in the current code paths, CRIX instead tries to identify missing-check bugs in the peer code paths. To this end, CRIX constructs peer slices that share similar semantics and contexts with the current code path checking the critical variable. To find substantial peer slices, given a critical variable, CRIX identifies the sources and uses of the critical variable, and employs data-flow analysis to find slices for each source and each use (see §4.3).

A slice of a source or a use of a critical variable may or may not contain a security check. A naive approach is to iden-

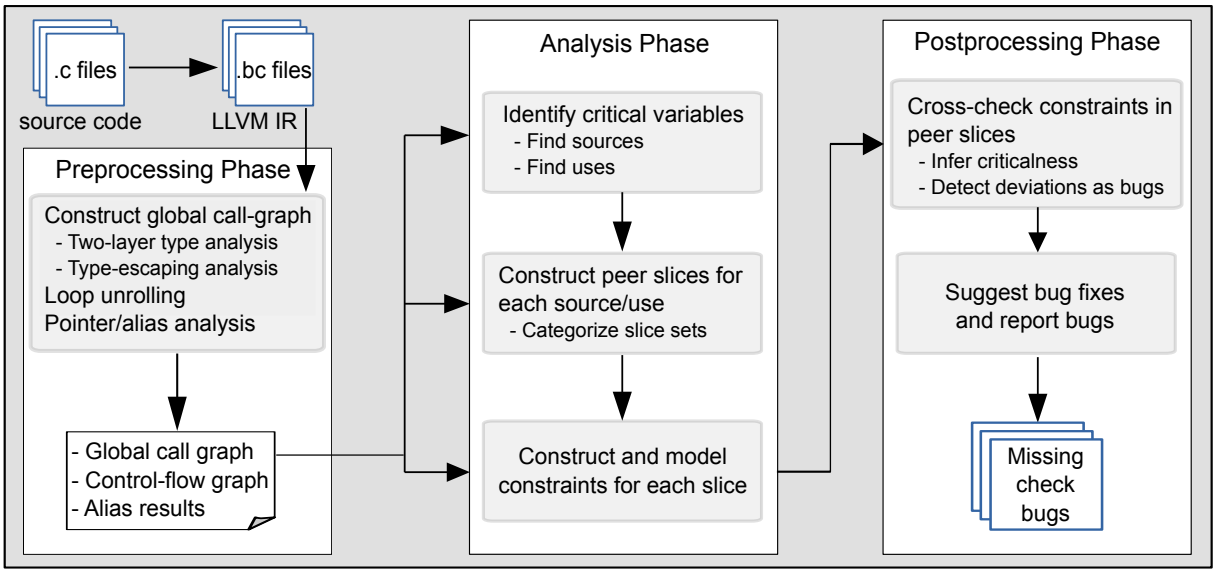

Figure 4: The overview of CRIX. CRIX has three phases. It takes as input LLVM IR and produces missing-check bug reports.

tify any slices that do not have a security check as potential missing-check bugs. This will however introduce significant false positives because (1) the source or the use may not be very "critical"; (2) even if security checks are present in some slices, they may not be semantically equivalent. To address this problem, CRIX first extracts the constraints from the conditional statements in the peer slices and models them in a special way (see §4.4) that can both preserve the semantics and facilitate the following bug detection.

With the modeled constraints extracted from conditional statements, in the last phase, CRIX cross-checks (statistical analysis) them to infer the "criticalness" of the source or the use based on how common the constraints are, i.e., how frequently the source or use is checked in its peer slices. If the criticalness is significant, not having a constraint would be identified as a deviation, and a slice that does not have the constraint would be identified as a potential missing-check bug. In the end, CRIX suggests bug fixes based on the constraints in the peer slices and reports the details for further manual confirmation.

4 Design of CRIX

In this section, we present the design of the key techniques in CRIX, including the identification of indirect-call targets, construction of peer slices, construction and modeling of constraints, and statistical analysis of constraints for reporting missing-check cases. Other techniques such as alias analysis and loop unrolling will be presented in the implementation section (§5).

4.1 Identifying Targets of Indirect Calls

A precise call graph serves as a foundation for a variety of program analyses and security defense mechanisms. In particular, any inter-procedural data- and control-flow analysis requires a precise call graph. Control-flow integrity (CFI) [1, 9, 10, 25] and software debloating [28] techniques also require a precise call graph. Unfortunately, in large programs, constructing a precise call graph is an open problem in general because of the challenge of finding the targets of indirect calls. At compilation time, it is hard to know which address-taken functions would be valid targets of an indirect call.

Existing approaches for finding the targets of indirect calls can be classified into two categories: pointer analysis [2, 3, 8, 22, 36, 37] and type analysis [4, 9, 25, 40, 42]. Pointer analysis–based approaches aim to find the point-to relationships between dereferenced function pointers and address-taken functions. Such approaches have fundamental limitations. While unsound pointer analysis will miss valid function targets, sound pointer analysis often introduces a large number of false positives—many unrelated functions are included as potential targets of an indirect call. Further, the pointer analysis *itself* requires a precise call graph. Whenever the pointer analysis encounters an indirect call, an expensive recursive analysis must be employed to find the targets.

Due to the limitations with pointer analysis–based approaches, recent CFI research opted for type analysis. Type analysis–based approaches try to match the number and types of arguments of an address-taken function with the ones of an indirect call. Matched functions are considered potential targets of the indirect call. Such approaches have been used in practice. For example, LLVM-CFI [4] employs such a type analysis. Type analysis–based approaches are conservative in

that all possible targets are included as long as function-type casting, which is rare, is handled properly [25]. However, they tend to suffer from false positives—many unrelated functions are included as valid targets. This will cause significant inaccuracy in the following data-flow analysis. The problem becomes even more critical in CRIX because the construction of peer slices heavily relies on a precise call graph.

To the best of our knowledge, the hybrid approach proposed by Ge et al. [10] for finding indirect-call targets in OS kernels is the most precise one. It employs both taint analysis and type analysis to find the targets. Specifically, it first taint-tracks the propagations of function pointers to identify indirect-call targets. Moreover, for function pointers stored in `struct`-type objects, because the function pointers should typically be loaded from the objects of the same `struct` type, the approach uses the `struct` type to further restrict the indirect-call targets. To avoid false negatives, the approach has two assumptions: (1) the only allowed operation on a function pointer is assignment, and (2) there exists no data pointer to a function pointer. The approach uses static taint analysis to detect and report violations which will be fixed manually. In addition to the hybrid analysis, the approach also analyzes assembly code, which further restricts the indirect-call targets.

4.1.1 Two-Layer Type Analysis

To improve the existing type analysis–based approaches in finding indirect-call targets, we propose *two-layer type analysis*, which aims to dramatically refine the targets produced by previous type analyses. The first-layer type analysis uses function types to restrict indirect-call targets. The second-layer type analysis instead uses `struct` type to further restrict the targets, which is based on a similar observation as in the approach proposed by Ge et al. [10]. Specifically, in large systems such as OS kernels, the majority of taken addresses (e.g., 88% for the Linux kernel, according to our study in §6) of functions are first stored to a function-pointer field of a `struct`, and later, to dereference the addresses in indirect calls, they must be loaded from the `struct`. In LLVM IR, the type information of the `struct` in both store and load operations is present. Intuitively, in these cases, function addresses that are *never* stored in the specific `struct` will not be valid targets of the indirect calls that load the function addresses from the `struct`. This way, by further matching the `struct` types in the store and load operations, we can further refine the indirect-call targets. 12% of function addresses in the Linux kernel are not stored to `struct`. A common example in the Linux kernel is that a function address is stored to a function-pointer variable which is further used as an argument of another function. Indirect calls dereferencing these function pointers will not benefit from the second-layer struct-type matching.

Figure 5 shows an example, in which the addresses of functions `adp5589_reg` and `adp5585_reg` are stored in the `reg` field of a `struct` with type `adp_constants`, in line 10 and 16, respectively. Later on, the addresses are loaded from the field of the `struct` of the same type and dereferenced at line 4. Our two-layer type analysis finds exactly only *two* targets for the indirect call because there are no any other functions whose addresses are ever stored to the field of the `struct` type. In comparison, since the indirect call has only one argument of a basic type, traditional one-layer type analysis matches 20 functions as targets for the indirect call, 18 of which are false positives.

A struct may have multiple fields that hold function pointers. To further improve the analysis accuracy, our type analysis is field-sensitive. That is, it recognizes which field is holding the particular function pointer, by analyzing the offset of the field in the data struct. In some rare cases, when the offset is undecidable because the indices are non-constant, we roll back the analysis to be field-insensitive.

```
1  /* drivers/input/keyboard/adp5589-keys.c */
2  static int adp5589_gpio_add(...) {
3      /* Indirect call: "kpad->var" is of type "adp_constants" */
4      kpad->var->reg(ADP5589_GPIO_DIRECTION_A);
5  }
6
7  unsigned char adp5589_reg(unsigned char reg)
8  static const struct adp_constants const_adp5589 = {
9      // address of "adp5589_reg" assigned to the field "reg"
10   .reg      = adp5589_reg,
11 };
12
13 unsigned char adp5585_reg(unsigned char reg)
14 static const struct adp_constants const_adp5585 = {
15     // address of "adp5585_reg" assigned to the field "reg"
16   .reg      = adp5585_reg,
17 };
```

Figure 5: An example of how a function pointer is stored to and later loaded from a field of a `struct`.

4.1.2 Type-Escaping Analysis for False Negatives

Our two-layer type analysis is sound as long as the `struct` types holding function addresses do not escape—we cannot decide what function addresses a `struct` can hold. When a `struct`, say `structA`, has escaped, a function address stored to a different `struct`, say `structB`, can be loaded from the memory with `structA`; however, in this case, the function address will be missed by the type analysis because we cannot find that the function address is ever stored to `structA` but only `structB`. Such escaping cases exist when (1) the `struct` holding the function addresses is cast to or from a different type; (2) the function-pointer field of `struct` is stored to with a value of a different type (e.g., `unsigned long`). These cases may make the function addresses a `struct` can hold *undecidable*.

To handle this problem, we use conservative type analysis to find all store and casting operations and analyze the types in the sources and destinations based on the aforementioned criteria for deciding escaping cases. When an escaped type

is found, we conservatively discard the type in our two-layer type analysis. That is, if the function pointer of an indirect call is loaded from an escaped type, we use only one-layer type analysis for this indirect call. This way, we ensure that our two-layer type analysis does not introduce extra false negatives to existing one-layer type analysis.

Although CRIX shares the similar insight into further restricting indirect-call targets with the approach proposed by Ge et al. [10], CRIX differentiates itself from the approach. CRIX employs a two-layer design that allows the type analysis to be elastic. Whenever the second-layer type analysis fails, CRIX falls back to the first-layer type analysis. Second, the escaping analysis conservatively finds and discards invalid types to ensure the soundness.

4.2 Identifying Critical Variables

System software has a large number of variables. Conservatively checking all of them is not only unscalable but also generates an overwhelming number of false reports. Intuitively, important variables are often protected with security checks. We say that a variable is a (potential) critical variable if it is validated in a security check. By identifying security checks and their targets, we can identify critical variables. Note that a critical variable has different levels of *criticalness*. As will be shown §4.4, the criticalness is inferred based on check ratio of the occurrences of the critical variable. In this section, we first focus on identifying critical variables.

4.2.1 Identifying Security Checks for Critical Variables

Since we define validated variables in security checks as critical variables, CRIX first identifies security checks using a similar approach proposed in LRSan [43]. Specifically, checking failures typically require failure handling which has clear patterns: returning an error code or calling an error-handling function. We say that an `if` statement is a security check if its two branches satisfy the following two conditions: (1) one branch handles a checking failure, and (2) the other branch continues the normal execution. Note that an `if` statement whose two branches both handle checking failures is not a security check. Therefore, the key step to identify security checks is to determine whether the branches have the failure-handling patterns. Two typical failure-handling primitives are returning an error code and calling an error-handling function. Since LRSan supports only error-returning cases, we extend the idea by supporting error-handling functions.

System software such as the Linux kernel has a small number of basic error-handling functions. Such functions are often critical and implemented in assembly. For example, `BUG()`, `panic()`, and `dump_stack()` in Unix-like OS kernels are functions for handling unrecoverable errors. Moreover, functions such as `pr_err()` and `dev_err()` are used for reporting error messages, which have clear patterns. Specifically, such functions typically have a name or an argument with a severity level (e.g., `KERN_ERR`, `KERN_CRIT`, and `KERN_EMERG`). Moreover, such functions take a variable number of parameters. Detecting these patterns is straightforward for a static analysis tool. To ensure that our heuristic-based approach reports correct error-handling functions, we manually investigated the results and filter out false-positive cases. In total, we found 531 error-handling functions (available in the code repository). In comparison, while LRSan reports only 131K security checks, CRIX reports 308K security checks. Once we identify security checks, we extract the checked targets as critical variables.

4.2.2 Identifying Sources and Uses of Critical Variables

In the next step, CRIX collects the sources and uses of the critical variables (i.e., checked variables). It is important to identify sources (where a critical variable propagates from) and uses (where a critical variable is used) of critical variables for two reasons. First, criticalness of a variable can propagate. When a critical variable is moved to another variable, the destination variable also becomes critical. By identifying the sources and uses, we can identify families of critical variables that propagate from the same sources or propagate to the same uses. Second, by identifying a family of critical variables, we can analyze how frequently they are checked, which is used to infer the criticalness of a source or a use. We realize the identification of sources and uses through a standard *inter-procedural* data-flow analysis—backward analysis for identifying sources and forward analysis for identifying uses. The inter-procedural data-flow analysis uses the following definitions to identify sources and uses.

Definition of sources. If a value is never critical, we do not need to include it for further analysis. Therefore, we include only potentially-critical values as sources. The inter-procedural backward data-flow analysis collects the following variables as sources.

- Constants. Constants such as error codes are critical.
- Return values and parameters of certain functions. Input functions (e.g., `copy_from_user` and `get_user`) obtain inputs from the external entities, which are untrusted. In addition, functions implemented as handwritten assembly often perform critical operations. We include the corresponding parameters or return values of such functions as sources.
- Global variables. Global variables may contain critical values that may propagate to the whole program.
- Others. When CRIX cannot find a predecessor instruction, the current values are marked as sources.

Note that we do not include allocations as a source because they become critical only when critical values are written to the allocated memory.

Definition of uses. Further, the following operations are defined as potentially critical uses of critical variables.

- Pointer dereference. Any pointer dereferencing operation is a critical use of the pointer variable.
- Indexing in memory accesses. Using the offset variable in memory access is also a critical use.
- Binary operations. We conservatively treat binary operations such as arithmetic division as critical uses.
- Functions calls. When none of the above is found, we take the closest function call that takes the critical variable as a parameter as a critical use.
- None. If none of the above is found, we deem that this critical variable does not have any use. This is common for critical variables that are error codes.

Algorithm 1: Collect sources and uses of critical variables

```
collect_src_use_interprocedural(CVSet, FuncSet);
  Input: CVSet: Critical variables, i.e., identified checked variables;
         FuncSet: Input and assembly functions, collected in the
         pre-processing phase
  Output: SrcSet: Potentially critical sources of critical variables;
          UseSet: Potentially critical uses of critical variables

SrcSet ← UseSet ← ∅;
BackupSet ← CVSet;
// Collect sources
while Is_Not_Empty(CVSet) do
    CV ← pop top element from CVSet;
    if CV is Constant and CV is ErrorCode then
        SrcSet ← {CV} ⋃ SrcSet;
    else if CV is Global variable then
        SrcSet ← {CV} ⋃ SrcSet;
    else if CV is return value or param. of a function in FuncSet
      then
        SrcSet ← {CV} ⋃ SrcSet;
    else if CV has no parents (predecessors) then
        SrcSet ← {CV} ⋃ SrcSet;
    else
        Parent ← Predecessor of CV, via Backward Analysis;
        CVSet ← CVSet ⋃ Parent;
    end
end
// Collect uses
CVSet ← BackupSet;
while Is_Not_Empty(CVSet) do
    CV ← pop top element from CVSet;
    CVUseSet ← Forwardly collect immediate uses of CV;
    for Use ∈ CVUseSet do
        if Use is a pointer dereference or memory access then
            UseSet ← {Use} ⋃ UseSet;
        else if Use is a binary operation then
            UseSet ← {Use} ⋃ UseSet;
        else if Use is a parameter of function then
            UseSet ← {Use} ⋃ UseSet;
        else
            CVSet ← CVSet ⋃ {Use} ;
        end
    end
end
return SrcSet, UseSet;
```

Algorithm for identifying sources and uses. Based on the definition of sources and uses, the algorithm presented in Algorithm 1 collects all potentially critical sources and uses. The algorithm takes as input the set of critical variables (`CVSet`), and set of functions (`FuncSet`). `CVSet` are the checked variable extracted from a security check, and `FuncSet` is the set of pre-collected input functions (e.g., `copy_from_user`) and assembly functions. `FuncSet` is collected in the pre-processing phase of the CRIX, concurrently with security-check identification, as will be shown in §5. The algorithm then produces two sets as the output: `SrcSet` and `UseSet`, the source and use sets, respectively. `CVUseSet` contains the immediate and forward uses of the current `CV`, which are returned by LLVM's `value.users()` function. As shown in Algorithm 1, the analysis is recursive and inter-procedural. Note that the algorithm is used to collect potentially critical sources and uses, but not to infer criticalness. Criticalness is instead inferred by measuring how frequently a critical variable is checked before being used, as will be shown in §4.4.

4.3 Constructing Peer Slices

At this step, we have the sources and uses of critical variables. Seemingly, we can construct slices for the critical variables forwardly from their sources and backwardly from their uses, and cross-check the slices to find check deviations as potential missing-check cases. Such a naive approach will suffer from at least two problems. First, the slicing will easily lead to path explosion [15] given the complexity of OS kernels. Second, if slices do not share similar semantics and contexts, we cannot effectively detect missing-check bugs because missing a check in an unrelated slice does not necessarily indicate a potential bug. Consequently, such an approach will lead to significant false positives.

To solve these problems, for a source or a use, we must construct its peer slices. Such peer-slice construction should satisfy two requirements: (1) the construction should yield sufficient peer slices to enable cross-checking; (2) the peer slices should share similar semantics and contexts. Given a control-flow graph, we observed that `call` (both direct and indirect) and `return` instructions often generate peer paths. In particular, for sources, indirect calls and return instructions often have substantial targets. As the example shown in Figure 5, indirect call `pad->var->reg()` serves as a dispatcher that may target multiple semantically similar callee functions (e.g., `adp5589_reg` and `adp5585_reg`). Since the arguments in the callee functions all come from the same caller, they also share the similar contexts. For uses, when the used critical variable comes from an argument of the current function, direct calls to the function also generate substantial edges from the callers to the function (callee). Since the arguments passed from various callers to the same callee function, they are used as similar semantics in similar contexts.

Figure 6 illustrates how we find different classes of peer paths for sources and uses. For each critical-variable source, we perform forward data-flow analysis for it. When encoun-

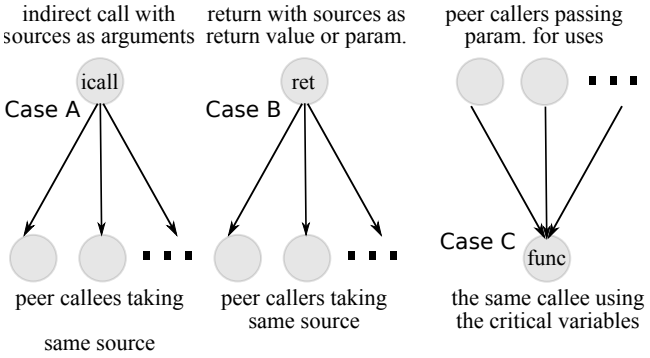

Figure 6: Different cases generating peer paths. `icall` is indirect call; `func` is the callee taking critical variables from peer callers.

tering an indirect call that takes the source as a parameter, we collect all the indirect-call callees as a set of peer paths. Similarly, when encountering a return instruction, we analyze whether the critical variable is returned or written into the memory pointed to by an argument (in this case, the critical variable may be further used in the callers through a pointer parameter). If so, we collect callers (starting from the next instruction following the call) as a set of peer paths. Our analysis is recursive. That is, the forward data-flow analysis continues to find more sets of peer paths until the end of the propagation of the critical variable or a critical use of the variable is found. For uses, we instead perform backward data-flow analysis from a use of a critical variable. If the critical variable comes from an argument of the current function, all callers of the function are collected as peer paths. The backward analysis is also recursive and ends until the source of the critical variable is found. Since a peer path may further contain multiple sub-paths, we use a simple `BFS` algorithm to flatten all sub-paths. Therefore, each peer path can be viewed as a single path. Finally, we construct peer slices by slicing the peer paths. The slicing ends at a conditional statement or the end of the path. Therefore, each slice has at most one conditional statement. Note that ending at the closest conditional statement would not cause false negatives because the slices sets are collected in a recursive manner, and our detection in §4.4.2 will cross-check each peer set.

For each critical-variable source and use, the peer-slice construction produces multiple sets of peer slices and categorizes them into four classes, each corresponds to a case in Figure 6.

- `Source-Ret` corresponds to `case B`. A critical variable is returned as the return value to multiple peer callers.
- `Source-Param` also corresponds to `case B`. However, in this case, a critical variable is "returned" an output parameter to multiple peer callers.
- `Source-Arg` corresponds to `case A`. A critical variable is passed to peer callees through an indirect call.
- `Use-Param` corresponds to `case C`. A critical variable used in a function is passed in from multiple peer callers.

4.4 Constructing and Cross-Checking Check Constraints

Until now, CRIX has produced multiple sets of peer slices of different classes for each critical-variable source and use. Each slice may or may not contain a conditional statement. The next step of CRIX is to cross-check the slices to detect deviations in the absence of security checks as potential missing-check cases. We choose to cross-check conditional statements instead of security checks (a subset of conditional statements) in this step for two reasons. First, the security-check identification part in CRIX have false negatives and may not identify all security checks; cross-checking security checks only may have significant false negatives because deviations can be normalized. Second, although cross-checking all conditional statements may introduce false positives, our fine-grained modeling for conditional statements can mitigate this issue.

A simple approach to cross-check slices for deviations is to treat conditional statements equally and quickly find deviating slices that do not have any conditional statement. Such coarse-grained analysis may have false negatives because conditional statements may have completely different semantics, and having a conditional statement does not mean the slice has checked the source or use. On the other hand, exactly comparing concrete values in conditional statements would be too restrictive, leading to false positives. For example, when two slices have `if (len < 8)` and `if (len < 16)`, respectively, treating them as different checks is too aggressive because both of them indeed enforce length checks. To avoid these problems, we must "qualitatively" understand the semantics of conditional statements in the slices. To this end, we propose to construct and model constraints from conditional statements. Note that the modeling focuses on the semantics of conditional statements, which does not consider their positions in the slices.

4.4.1 Modeling Conditional Statements as Constraints

As described in §4.3, a slice is flattened as a single code path using BFS, and a slice has at most one conditional statement. The goal of this step is to answer what classes of semantics a conditional statement has, with a proper granularity. We thus use two empirical rules to model the conditional statements based on the semantics of typical conditions and comparison operators. The modeled conditional statements will be cross-checked for missing-check bugs, as shown in §4.4.2.

1. If the conditional statement checks the return value of a function call, we identify the function's signature as the constraint. For example, if a variable is checked in a conditional statement, `if (IS_ERR(ret))`, we model the constraint as "`IS_ERR(int)`". This is, the slice uses `IS_ERR()` to check the source or use.
2. Otherwise, we model the conditional statement as "`<opcode_type, operand_type>`", where `opcode_type`

represents the type of the comparison, such as `eq`, `ne`, and `lt`; and `operand_type` represents the type of the condition operand. The type can be `var` (a variable), `zero`, `positive` constant, and `negative` constant. For example, if a condition statement is `if (len < 8)`, the constraint will be modeled as "`lt positive`".

4.4.2 Detecting Deviations as Potential Bugs

With the modeled constraints for all slices in a set, we cross-check them to find deviations. The idea of the detection is to calculate the `relative frequency (RF)` [18] for each constraint in the set. Since different constraints have different frequency distributions, we calculate the RF for each constraint in the set separately. More specifically, given a constraint, we define N_{nc} as the number of slices that do not have the constraint, and define N_t as the total number of slices in the set. With these two numbers, the RF is defined as in Equation 1.

$$RF = \frac{N_{nc}}{N_t} \tag{1}$$

The detection works as follows. Given a constraint in a peer-slice set, the detection counts how many slices do not have this *particular* constraint. Note that a slice that has a different constraint will also be counted. The count serves as N_{nc}. Since N_t is the total number of slices in the set, we can quickly obtain it and calculate the RF for the given constraint. If the RF is very small, i.e., most slices have the constraint, the detection reports slices that do not have the constraint as potential missing-check cases. A slice set may have multiple constraints, and the detection will go through the steps for each constraint in the slice set.

5 Implementation

We have implemented CRIX as multiple passes on top of LLVM, including a pass for constructing call graph and unrolling loops, a pass for finding security checks and critical variables, and a pass for detecting and reporting missing-check cases. CRIX's source code contains 4.5K lines of C++ code. The rest of the section describes some interesting implementation details in each phase.

5.1 Preprocessing Phase

Disabling inlining and IR pruning. To facilitate peer-slice construction, we aim to preserve callsites as much as possible. To this end, we chose to disable inlining by modifying `Clang`. A side effect of disabling inlining is that *inline* functions defined in header files will be copied to each module that uses them, leading to significant redundancy in LLVM IR. To prune the IR, we leverage debugging information to map the functions to its source code. This way, we can figure out multiple functions in IR share the same source code, and if so, we keep only one copy in the IR and discard all other copies. The pruning strategy reduces the original size of IR by approximately 30%.

Identifying indirect-call targets. To realize the two-layer type analysis, we first identify all store operations (either a store instruction in LLVM or a struct initializer) that assign a function address to a variable. We then analyze the type of the *memory* holding the variable. At this step, our analysis is conservative: the variable must be loaded from a pointer, and the pointer must be pointing to a field of a data structure. That is, the pointer must be a `GetElementPtrInst` in LLVM. With the type information in LLVM IR, we can then extract the base struct type from the pointer. Note that, we do not recursively find the struct type; if the pointer is not `GetElementPtrInst`, or the base type of the `GetElementPtrInst` is not a `struct` or is an aggregated type (e.g., `union`), we stop the analysis for the particular function address and roll back to the traditional one-layer type analysis. The filed-sensitive analysis is realized by analyzing the indices in the `GetElementPtrInst` which includes the index of the accessed field into the base type. The process of this step goes through all address-taken functions in all modules. The output of this step is a map from the hash of the type to the function addresses.

A challenge in implementing the two-layer type analysis to conservatively capture escaping types. As described in §4.1.2, we have a conservative policy to identify escaping types. To implement the type-escaping analysis, we analyze the operand types in cast and store operations (both instructions and global static initializers). If the operand types satisfy the policy, we identify them as escaping types.

After that, we match the second layer type for indirect calls. Similarly, we analyze the type of the *memory* holding the function pointer (address) in the same way—analyzing the corresponding `GetElementPtrInst`. By querying the map, we can find the matched functions for the indirect call. If we cannot find a match, we again roll back to the one-layer type analysis for the indirect call.

Unrolling loops. To avoid path explosion, we chose to unroll loops by treating `for` and `while` statements as `if` statements, which is a common strategy used in practice [45]. A loop has two special basic components: header block, latch block. A header block is the entrance node for a loop; a latch block contains an edge back to the header block. In order to unroll loops, we delete the back edge and add a new edge from the latch block and the successor block of the loop.

Pointer analysis. We perform points-to analysis for each pointer to a memory location within a function, relying on LLVM's AliasAnalysis infrastructure. The `MayAlias` results conservatively include pointers that may refer to the same object; two pointers referring to different fields of an object may also be included as aliases. To refine the results, we perform field-sensitive data-flow analysis for each pointer that is ever used in memory load/store or function calls as

parameters. Pointers that are validated to refer to different fields are excluded from the `MayAlias` results. A second issue with the points-to analysis is its significant runtime overhead, and we observed that this is mainly caused by a small number of objects that have a large number of pointers. We mitigate the problem by limiting the maximum number of pointers an object can alias simultaneously. By setting the number to 1000, our results showed that only 23 functions in the Linux kernel have aliased memory pointers with size greater than this limit. After applying the two improvements, the running time for points-to analysis is reduced from 103 minutes to only 24 minutes, and the average number of alias pointers of a object is reduced by 65%.

5.2 Analysis Phase

Modeling input functions and collecting assembly functions. In CRIX, specific return values and parameters of input functions and assembly functions are defined as sources (§4.2.2). As such, we need to collect a set of such functions. We define a function as input function if it may fetch data from outside. For example, `copy_from_user(dst, src, size)` copies the content from user-space memory `src` into the kernel-space memory `dst`. In total, we empirically collected 36 input functions (Table 3). We also specified which parameter or if the return value of these functions holds the inputs. Similarly, the kernel contains lots of assembly code as optimizations for performance reasons. Such functions are typically critical. Since LLVM does not support analysis of assembly code, we also model the assembly functions and treat them as sources. Identification of assembly functions is realized by scanning through LLVM IR files for `isa<InlineAsm>` instructions.

5.3 Postprocessing Phase

Selecting threshold for relative frequency. Missing checks within the Linux kernel are identified using various strategies described in §4. A case in a peer-slice set that has low relative frequency will be reported as a potential missing-check bug. The relative frequency is the ratio of occurrences of a constraint or "non-constraint" to the size of the peer-slice set. A uniform threshold for different categories might skew the results in favor of a particular type of bugs. To solve this challenge, we provided the relative frequency field as a tuning parameter and tested the results on various runs for various categories. We observed that the relative frequency works best between [0.1, 0.15] to detect sufficient missing-check bugs with reasonably low false reports, for all the categories.

Generating bug-fixing suggestions. Peer slices of the same critical variable can reveal many interesting details about the implementation. For each peer, we are able to reason about the constraints of the critical variable. Since we have constraints for each of the peer slices, one can suggest a possible security check in a possible location for missing-check cases. Statistically analyzing the "suggestions" returns us a reasonable bug fix. As such, CRIX always reports the most common suggestion to facilitate bug fixing. The report includes the most common constraint and which function the constraint should be applied to.

Bug Reporting. After collecting the constraints from the peer slices and using a user-defined relative frequency, we rank the missing-check output based on the relative frequency. we format the report to output the line contains the relative frequency, the Linux source code, the module containing the code, the number of times security check was checked, times missed among the peers, and most importantly, the bug-fixing suggestion. As expected, reported cases in the top of the ranking are more likely to be true bugs.

6 Evaluation

We extensively evaluate the scalability and effectiveness of CRIX using the Linux kernel. We also evaluate the effectiveness of our two-layer type analysis. 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. We tested the bug detection efficiency of CRIX, on the Linux kernel version 4.20.0-rc5 with the top git commit number `b72f711a4efa`, the latest patch as on Dec 6, 2018. Using the `allyessconfig`, we generated 17,343 LLVM IR bitcode files to cover as many modules as possible.

6.1 Precision in Finding Indirect-Call Targets

Results. In total, out of 57,299 indirect calls, 45,840 (80%) enjoyed our two-layer type analysis. 5,019 (8.8%) indirect calls suffer from type escaping thus disqualify the two-layer type analysis. Others indirect calls do not load function pointers from a `struct` thus do not trigger the two-layer type analysis. The high percentage confirms our observation that most function pointers are stored to and loaded from memory through data `struct`. We then calculate the average number of targets for an indirect call before and after applying our two-layer type analysis. The results show that the average number over all indirect calls for traditional type analysis is 134 while it is only 33 for our two-layer type analysis. We further calculate the average numbers over indirect calls that can benefit the two-layer type analysis. The results show that the average target number is 129 and 9 (i.e., 7%) before and after using our two-layer type analysis, respectively, which confirms that the analysis can dramatically refine the indirect-call targets.

Measuring the false positives of indirect-call targets is a challenging problem because of the complexity of pointer propagation and point-to relationships. Existing CFI tech-

niques use the average number of targets to represent the accuracy of target refinement. Given that CRIX reports only an average number of 9 for indirect calls that benefited from the two-layer type analysis, we expect the false-positive rate of our analysis to be low. In comparison, the hybrid approach proposed by Ge et al. [10] reports an average number of 6.64 for indirect calls in FreeBSD, and, the number is calculated over all indirect calls. We believe that the accuracy of the approach benefits from the combination of taint analysis and type analysis.

6.2 Analysis Performance and Numbers

CRIX completed the analyses of the kernel for missing-check cases in 64 minutes, of which pointer analysis required 24 minutes and the remaining analysis to identify and report missing-check cases required 28 minutes. By running CRIX over the whole kernel, with a threshold of `0.15`, the output contained 308K security checks from 1,028K conditional statements, and reported 804 cases.

6.3 Bug Findings

Table 2 presents the bug detection statistics of CRIX, running on the entire Linux kernel with a constant relative frequency of 0.15, across categories. We used a fixed number for relative frequency to avoid inconsistencies while comparing similar bugs across the various categories. CRIX reported 804 potential bugs and manual analysis confirmed 278 new bugs. To manually analyze all the bugs, it took three researchers, a total of 36 man-hours. We found that the cross-checking results over peer slices can significantly relieve the manual analysis by suggesting how and why peers enforce the security checks. The manual effort was mainly spent in checking if the critical variable is actually checked because the check may have been missed by CRIX due to issues such as aliasing. In most cases, the "suggested" source-check or check-use chains are across one or two functions, so the manual analysis overall is straightforward.

We submitted patches for all the bugs. Linux maintainers accepted 151 of the submitted patches to be applied to the latest Linux version or future releases. Maintainers confirmed 99 patches within a week of submission confirming the criticalness of fixing missing-check bugs. Figure 1 shows an example of the new bugs found by CRIX, which can cause multiple security issues such as NULL-pointer dereferencing. A detailed list of all the bugs is available in Table 4 and in Table 5, in the Appendix section. During our interaction with the maintainers, we not only fixed missing-check bugs determined by CRIX, but also fixed some other relevant bugs present in the error paths of security checks including but not limited to missing/incorrect error handling, missing resource releases, use-after-free, and dead code.

Further 76 bugs are included in more than one bug category. That is, these 76 bugs were detected twice, once each while generating the source and use constraints. However, these duplicates are within the chosen 804 cases, used to evaluate CRIX. Accounting for the duplicate bug reporting, the false-positive rate of CRIX is 65%. We believe this is an acceptable number for critical software such as OS kernels.

The distribution of bugs is heavily skewed towards driver code. The report showed 195 bugs in the driver modules and at least 27 driver modules had more than one missing-check. These bugs reinforce previous research studies that the driver code is indeed buggy as well as confirm the effectiveness of CRIX in detecting new missing checks. Second, we also computed the latent period of the detected bugs and the average time between the initial patch and detection is 1,675 days or approximately 4 years and 7 months. A significant observation is 27 out of these 278 bugs have a latent period of greater than 10 years and 6 patches' latent period is greater than 13 years.

The third interesting finding of our bugs involves the type of bugs. A total of 79 bugs involve memory allocation on the heap. Linux developers strictly maintain that every pointer returned by an `alloc`-like function be checked for emptiness. Interestingly, CRIX identified 11, 5, 11, and 12 calls of (`kzalloc, kmalloc, kcalloc` and `kmemdup`) respectively; all missing a check on the pointer to the allocated memory for emptiness. All these bugs can crash a system while dereferencing the NULL pointer, as well as provide an attack vector to launch a denial of service attack by unauthorized users. The numerous missing-check bugs confirm the effectiveness of CRIX in identifying security vulnerabilities.

One reason of concern in the output report is the duplication of bugs across various categories. CRIX performs backward data-flow analysis from use, and a forward data-flow analysis from source to identify missing-check bugs in various categories. With a constant relative frequency, a true missing-check bug will often be reported in both directions. To simplify our analysis, we ran CRIX performing both analyses at the same time and then eliminated the duplicate records. While this action does not impact the accuracy of the system, we observed a non-trivial difference in ranking order of the bug, when evaluating each category individually.

6.4 False Positives

As presented in §6.3, CRIX has false positives. We have investigated the causes of false positives. In this section, we present the main classes of causes.

Inaccurate points-to analysis. Pointer analysis [14] is a hard problem. CRIX's data-flow analysis engine generally relies on the Alias Analysis. However, the alias results provided by LLVM are often inaccurate. Al through we have refined the `MayAlias` results, there are still over 48% of false positives that are caused by the inaccuracy of pointer analysis. We will

Category	Example bug (related function)	Latent Period	R	A	C
Source-Ret	drivers/net/hyperv/netvsc_drv.c +1377 (kvmalloc_array)	4y 10m	449	300	156
Use-Param	net/ncsi/ncsi-netlink.c +253 (nla_nest_cancel)	2y 10m	247	150	115
Source-Param	drivers/gpu/drm/i810/i810_dma.c +307 (drm_legacy_ioremap)	4y 1m	83	42	4
Source-Arg	drivers/dma/ti/omap-dma.c +1056 (omap_dma_prep_dma_cyclic)	10y	25	8	3

Table 2: Bug detection statistics of CRIX on Linux kernel with relative frequency = 0.15. Columns R= bugs reported, A = Analyzed bugs, C = Confirmed bugs. The Latent Period is the average time differential for all confirmed bugs(C), within the category.

discuss potential improvements of pointer analysis in §7.

Inconsequential checks. While checks are necessary to guarantee the state of the kernel, programmers often ignore security checks in cases such as debugging code, failure-handling paths, driver-shutdown functions, resource cleanup functions, unlikely failures (e.g., `kmalloc` with `__GFP_NOFAIL`) or code that is already protected by synchronization primitives. Checks are redundant in these cases as an erroneous state has already existed or a valid state is guaranteed by the kernel. Such cases account for 25% of false positives.

Implicit checks. Programmers can reason about the state of the variable in an implicit way. For example, to test if an allocation of an object is successful, developers may use the object, without a security check, in a function, and use the return values of the function to test if the object was allocated successfully. In this case, although the object itself is never explicitly checked, the function call checks the object implicitly. Such cases contribute about 8% of the false positives. A potential solution to mitigating this problem is to maintain a list of "checker" functions.

Other causes. Besides these above-mentioned causes, false positives can also be caused by complex programmer logic, imprecise static analysis techniques, etc. All these account for the remaining 19% of the false positives.

6.5 False Negatives

CRIX provides a tuning parameter, the threshold of RF, while detecting missing-check cases. In other words, the threshold can influence the false-negative rate. In this section, we evaluate (1) the absolute false negatives that are missed when the RF threshold is set to 1; and (2) the relationship between the false-negative rate and the RF threshold.

In §2, we collected 119 missing-check bugs that have a clear security impact. To make the false-negative evaluation more robust, we collect 231 recently reported missing-check bugs in the Linux kernel based on its `Git` patch history. The patches containing the fixes of these 350 missing-check bugs are evaluated in our false-negative study. To reproduce these bugs, we revert the patches in the Linux kernel to the version used in our experiments.

By setting the RF threshold to 1, we determined 14 (4%) patches as absolute false negatives. Absolute false negatives are caused by two factors. First, the checked critical variables are not captured by CRIX because the error code or error-handling functions are not identified as part of a security check. Second, inaccurate pointer analysis identifies incorrect aliases for the critical variables. These aliases are mistakenly identified as valid security checks, bypassing the identification of actual missing-check bugs.

Second, we also evaluate the relationship between the threshold and false-negative rate, presented in Figure 7. We find that, as RF threshold increases, the false-negative rate decreases as prior false negatives are identified as missing-check bugs. We found that when the threshold is set to 0.13, the false-negative rate is 5% and reaches its elbow point. Further tuning the threshold has no impact on the false negative rate.

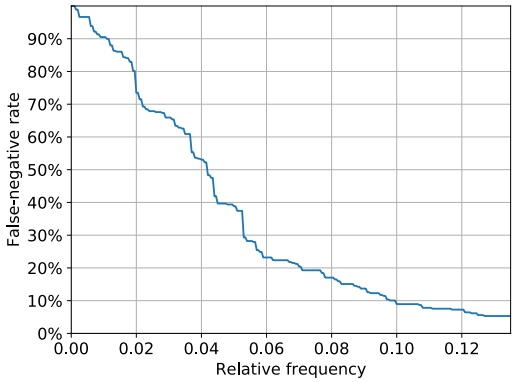

Figure 7: Relationship between the relative frequency (RF) threshold and the false-negative rate.

6.6 Portability

A program-specific component of CRIX is identifying security checks. Determining if a conditional statement is a security check, while scanning the kernel relies on the identification of error handling functions, and error codes. Identifying the error handling functions requires a limited amount of experience with the target code base. In Unix-like kernels, these functions share similar patterns, called "Single Unix Specification" [35], as presented in §4.2.1. All these kernels, also have a single global header file that defines the standard error codes.

Similarly, other kernels and programs like browsers also have corresponding header files containing the error codes.

Besides this step, the idea to generate an error control flow graph is generic to adapt to other systems. Once security checks, described in §4, are identified, the algorithm for identifying missing checks is easily adaptable to other software systems such as BSD kernels and C++ code base such as web browsers.

7 Discussion

Two-layer type analysis for more types. The current implementation of two-layer type analysis supports only `struct` type because it is the most commonly used type for memory holding function pointers. To further exploit the two-layer type analysis, we could extend it to support more types such as `array`, `global variable`, and `vector`. The type-escaping analysis in CRIX will ensure to eliminate false negatives when any type casting occurs or function pointers are moved across different types.

RF threshold. We discussed the false positives and false negatives of CRIX in §6. To balance false positives and false negatives, we suggest setting the RF threshold to a value between 0.1 and 0.15 for the Linux kernel. CRIX cross-checks peer slices to detect deviations. CRIX may have higher false-report rates in smaller target programs because they have small sets of peer slices.

Pointer analysis. Another major cause of false reports is the inaccuracy of alias analysis. To mitigate this problem, we intend to use Andersen pointer analysis [13] and Steensgaard pointer analysis [34] in the future. Given that, pointer analysis is used extensively in CRIX, we believe that this addition can significantly improve the overall accuracy.

Inconsequential checks. Besides alias analysis, the next major portion of false positives are due to programmer intended missing checks. Based on our interaction with Linux maintainers, we found that they are reluctant to fix missing-check cases in resource-release paths such as driver shutdown or state reset. Previous work by Saha et.al [31] proposed a pattern-based approach to find resource-release paths. As a potential solution, we may leverage the approach to filter out cases in resource-release paths and thus reduce the false positives in CRIX.

Determining exploitability and security impact of missing-check bugs. To automatically determine the exploitability of missing-check bugs, one can employ symbolic execution [29] and a theorem prover like Z3 [6] to generate inputs to trigger a missing-check bug. In addition, fuzzers can complement the limitations with symbolic execution. To automatically determine the security impact, one can analyze the uses of the checked variable to understand the potential security impact. For example, if a checked variable is used as the size variable in `memcpy()`, the potential impact can be memory corruption or information leak. For the identified new bugs, we found that more than half of them will cause Denial-of-Service, and quite a few of them will cause out-of-bound access, as shown in Table 4 and Table 5.

In general, automatically determining exploitability and security impact of a bug is a challenging research problem. A number of recent works [47] have investigated into this problem. If we can automatically decide the exploitability and security impact of a potential miss-check case, we can automatically confirm a missing-check bug/vulnerability and thus automatically eliminate false positives. We will leave such an analysis for future work.

8 Related Work

Missing-check detection. The most closely related works to CRIX are about missing-check detection. LRSan [43] detects lacking-recheck bugs, a subclass of missing-check bugs. CRIX detects general missing-check bugs that include lacking-recheck bugs. Juxta [23] detects semantic bugs using cross-checking between semantically equivalent implementations of file systems. Most bugs found by Juxta are missing-check bugs. CRIX can detect missing-check bugs in all subsystems in the OS kernels and do not require multiple implementations of a subsystem. Other works utilizing complementary implementation techniques to detect missing-check include Vanguard [32], Chucky [46], AutoISES [38], Rolecast [33], and MACE [24]. To the best of our knowledge, none of the tools are scalable to a system as large as the OS kernel nor have an equivalent technique to reason about the semantics and contexts of a critical variable.

Error-code propagation and handling. To detect missing-check bugs, CRIX relies on error-handling primitives to find critical variables. Techniques in error-code propagation and handling, within the Linux kernel, include EIO [12], Hector [31], and by Rubio-González et al.[30]. Similarly, APEx [17], ErrDoc [39], and EPEx [16] reason about the error-code propagation in open source SSL implementations, either automatically or via user definitions. Unlike CRIX, all the above systems target a limited range of error returning code specifications and thus have significant false negatives. Further, these techniques do not consider error-handling cases that do not return any error code. According to our study, such error-handling cases are common.

OS-kernel analysis. Given the complexity, analysis targeting the entire OS kernels is challenging. Recent advances on kernel analysis can be mainly categorized into kernel source-code analysis and static IR analysis. Smatch [5] and Coccinelle [27] find bugs in the Linux kernel. While Smatch [5] relies on syntax tree–based intra-procedural analysis to find simple bugs such as NULL-pointer dereferences. Coccinelle [27] performs code-pattern matching to find specified bugs. In comparison, CRIX leverages flow-sensitive, context-sensitive, and field-sensitive inter-procedural analyses to identify missing check bugs.

To benefit from rich analysis passes in LLVM, recently,

many tools analyze OS kernels on LLVM IR. K-Miner [11] improves the efficiency of data-flow analysis by partitioning the kernel code along separate execution paths stating from system-call entry points. Dr. Checker [21] is also a static data-flow analysis tool that identifies bugs in the drivers. While K-Miner and Dr. Checker serve as general bug detection tools, there are also some detection tools specialized for detecting a specific class of bugs in OS kernels. KINT [44] detects integer overflows using taint analysis; UniSan [20] detects information leaks caused by uninitialized data reads, also using taint analysis.

9 Conclusion

Missing-check bugs are a common cause of critical security vulnerabilities. In this paper, we have presented CRIX, a scalable and effective system for detecting missing-check bugs in OS kernels. CRIX's detection is semantic- and context-aware with an inter-procedural and context-, flow- and field-sensitive data-flow analysis engine. We realized the detection by proposing multiple new and general techniques. In particular, the two-layer type analysis can dramatically improve the precision in finding direct-call targets. The automated critical-variable inference narrows down the analysis to a very small scope, thus scaling expensive analyses to OS kernels. The peer-slice construction and constraint modeling for conditional statements enable semantic- and context-aware analysis. With these techniques, CRIX has reasonably low false-report rates and outstanding analysis performance. By applying CRIX to the Linux kernel, we found 278 new bugs and maintainers accepted 151 of our submitted patches. The evaluation results show that CRIX is scalable and effective in finding missing-check bugs in OS kernels.

10 Acknowledgment

We would like to thank our shepherd, Trent Jaeger, and the anonymous reviewers for their helpful suggestions and comments. We are also grateful to Stephen McCamant for providing valuable comments and to Linux maintainers for providing prompt feedback on patching bugs. 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.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[3] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[4] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[5] D. Carpenter. Smatch - the source matcher, 2009. http://smatch.sourceforge.

[6] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

[7] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[8] X. Fan, Y. Sui, X. Liao, and J. Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for c++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340. ACM, 2017.

[9] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 28–39. ACM, 2018.

[10] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 179–194, 2016.

[11] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[12] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *FAST*, volume 8, pages 1–16, 2008.

[13] B. Hardekopf and C. Lin. 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, June 2007.

[14] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM. ISBN 1-58113-413-4.

[15] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Path-sensitive backward slicing. In *International Static Analysis Symposium*, pages 231–247. Springer, 2012.

[16] S. Jana, Y. J. Kang, S. Roth, and B. Ray. Automatically detecting error handling bugs using error specifications. In *USENIX Security Symposium*, pages 345–362, 2016.

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

[18] J. F. Kenney and E. S. Keeping. Mathematics of statistics-part one. 1954.

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

[20] K. Lu, C. Song, T. Kim, and W. Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23rd*

ACM Conference on Computer and Communications Security (CCS), Vienna, Austria, Oct. 2016.

[21] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[22] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11(1): 7–26, Jan. 2004. ISSN 0928-8910.

[23] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. 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.

[24] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan. 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.

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

[26] NVD. National vulnerability database, 2019. https://nvd.nist.gov.

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

[28] A. Quach, A. Prakash, and L. K. Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[29] D. A. Ramos and D. Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[30] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *ACM Sigplan Notices*, volume 44, pages 270–280. ACM, 2009.

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

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

[33] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[34] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

[35] W. R. Stevens and S. A. Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2008.

[36] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[37] Y. Sui and J. Xue. Value-flow-based demand-driven pointer analysis for c and c++. *IEEE Transactions on Software Engineering*, 2018.

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

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

[40] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, pages 941–955, 2014.

[41] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 762–774. ACM, 2014.

[42] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 934–953. IEEE, 2016.

[43] W. Wang, K. Lu, and P. Yew. 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.

[44] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[45] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.

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

[47] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[48] S. Özkan. Common vulnerabilities and exposures details, 2019. https://www.cvedetails.com.

A Appendix

Data fetch functions	
copy_from_user	_copy_from_user
__copy_from_user	raw_copy_from_user
strncpy_from_user	_strncpy_from_user
__strncpy_from_user	strndup_user
__copy_from_user_inatomic	memdup_user
__copy_from_user_inatomic_nocache	copyin
__constant_copy_from_user	memdup_user_nul
rds_message_copy_from_user	__get_user
snd_trident_synth_copy_from_user	vmemdup_user
ivtv_buf_copy_from_user	copyin_str
iov_iter_copy_from_user_atomic	fusword
__generic_copy_from_user	copyin_nofault
__copy_from_user_eva	fuword
__arch_copy_from_user	fubyte
__copy_from_user_flushcache	fuswintr
__asm_copy_from_user	get_user
copy_from_user_toio	copy_from_user_page
copy_from_user_nmi	copy_from_user_proc

Table 3: List of input functions collected based on heuristics.

Subsystem	Filename	Line#	Impact	Category	Status	LP
net	gvf.c	511	reliability	P	C	2
x86	ghv_init.c	107	DoS	S	A	1
x86	gtlb_uv.c	2013	DoS	S	S	8
char	hpet.c	978	reliability	S	S	4
firmware	gdriver.c	711	DoS	S	C	1
gpio	gpio-exar.c	150	reliability	U	A	2
gpu	gkfd_crat.c	404	DoS	U	S	1
gpu	gi915_gpu_error.c	230	reliability	S	S	2
gpu	gradeon_display.c	679	reliability	S	C	2
gpu	gvkms_crtc.c	227	reliability	U	A	<1
hid	hid-logitech-hidpp.c	1954	reliability	S	A	3
iio	gmax9611.c	531	DoS	U	S	2
iio	gmxs-lradc-adc.c	466	DoS	U	A	2
iio	ghmc5843_i2c.c	62	reliability	S	A	4
iio	ghmc5843_spi.c	62	reliability	S	A	4
infiniband	gcm.c	1921	DoS	S	C	6
infiniband	gi40iw_cm.c	3257	DoS	S	A	2
infiniband	gi40iw_cm.c	3260	DoS	S	A	2
input	gpm8xxx-vibrator.c	198	DoS	P	S	2
isdn	ghfcpci.c	2034	reliability	S	A	10
isdn	ghfcsusb.c	265	DoS	S	A	10
isdn	gmISDNinfineon.c	716	DoS	S	A	9
leds	leds-pca9532.c	531	crash /DoS	S	A	2
media	gstv090x.c	1449	reliability	S	S	10
media	gstv090x.c	1452	reliability	S	S	10
media	gstv090x.c	1456	reliability	S	S	10
media	gstv090x.c	2229	reliability	S	S	5
media	gstv090x.c	2607	reliability	S	S	10
media	gstv090x.c	2913	reliability	S	S	10
media	gstv090x.c	2957	reliability	S	S	10
media	gstv090x.c	2975	reliability	S	S	10
media	gvpss.c	520	DoS	S	A	6
media	grcar-core.c	267	DoS	S	S	1
media	grenesas-ceu.c	1684	DoS	S	S	1
media	grga.c	894	memory leak	S	A	1
media	grga.c	896	memory leak	S	A	1
media	grga.c	910	reliability	S	A	1
media	grga.c	875	reliability	S	A	2
media	grga.c	915	use-after-free	S	A	1
media	gvideo-mux.c	400	DoS	U	A	1
media	gvideo-mux.c	402	DoS	S	A	1
media	gusbvision-core.c	2301	reliability	S	S	9
memstick	gms_block.c	2141	DoS	U	C	5
mfd	sm501.c	1145	DoS	S	A	1
mmc	gmmc_spi.c	821	concurrency	U	A	9
net	gmcp251x.c	963	reliability	S	S	3
net	glan9303-core.c	1081	system crash	S	S	1
net	glan9303-core.c	1074	system crash	S	S	<1
net	gpcnet_cs.c	1424	DoS	S	A	8
net	gpcnet_cs.c	290	DoS	S	A	8
net	glio_main.c	1194	DoS	S	A	2
net	glio_vf_main.c	1961	DoS	S	S	2
net	glio_vf_main.c	612	DoS	S	S	2
net	glio_core.c	1213	DoS	S	A	1
net	glio_core.c	1685	DoS	S	A	3
net	gnicvf_main.c	2264	DoS	S	A	1
net	gfmvj18x_cs.c	549	DoS	S	A	8
net	gfm10k_main.c	42	reliability	A	A	2
net	gen_rx.c	721	DoS	U	S	<1
net	gocelot_board.c	256	DoS	U	C	1
net	gqla3xxx.c	3888	system crash	U	A	12
net	gqlge_main.c	4682	system crash	S	A	2
net	gsh_eth.c	3133	reliability	U	A	5
net	gravb_main.c	1996	reliability	U	A	3
net	grocker_main.c	2799	DoS	S	A	1
net	gdwmac-dwc-qos-eth.c	487	DoS	S	A	2
net	gdwmac-sun8i.c	1150	system crash	S	A	2
net	gfjes_main.c	1254	concurrency	U	S	3
net	gfjes_main.c	1255	concurrency	S	S	3
net	gnetvsc_drv.c	1377	DoS	S	A	<1
net	gadf7242.c	1269	DoS	S	A	<1
net	gcore.c	645	reliability	S	A	2
net	gcore.c	646	reliability	S	A	2
net	gcore.c	653	reliability	S	A	2
net	gcore.c	662	reliability	S	A	1
net	gcore.c	689	reliability	S	A	2
net	gcore.c	714	reliability	S	A	2
net	gcfg80211.c	5368	DoS	S	A	6
net	gcfg80211.c	5384	DoS	S	A	6
net	g3945-mac.c	3405	reliability	U	S	7
net	g4965-mac.c	6241	reliability	U	S	7
net	gcmdevt.c	342	DoS	U	A	7
net	gray_cs.c	395	system crash	U	S	8
net	gray_cs.c	409	system crash	S	S	8
net	gray_cs.c	423	system crash	S	S	8
net	gbase.c	471	system crash	S	S	4
net	gfw_common.c	648	DoS	S	A	8
net	gfw.c	600	DoS	U	A	8
net	gfw_common.c	623	DoS	U	A	8
net	gfw.c	744	DoS	U	A	8
net	gfw.c	448	DoS	U	A	8
net	gfw.c	562	DoS	S	A	8
net	gfw.c	1623	DoS	U	A	8
net	gfw.c	1759	DoS	U	A	8
staging	gfw.c	745	DoS	U	A	8
net	gcmdevt.c	342	DoS	U	A	8
net	gqlcnic_ethtool.c	1050	DoS	U	A	8
net	grsi_91x_mac80211.c	199	DoS	S	A	5
net	grsi_91x_mac80211.c	208	DoS	S	A	5
net	gmain.c	347	DoS	S	A	6
nfc	gse.c	345	DoS	S	S	4
nvdimm	btt_devs.c	200	DoS	S	C	3
nvdimm	btt_devs.c	217	system crash	S	C	3
nvdimm	namespace_devs.c	2250	DoS	S	A	2
pci	gpci-tegra.c	1552	buffer overflow	S	S	1
pci	gpcie-rcar.c	931	buffer overflow	S	A	5
pci	gpcie-xilinx.c	343	buffer overflow	S	C	4
pci	gpci-epf-test.c	571	DoS	U	A	2
pinctrl	gpinctrl-baytrail.c	1711	DoS	U	A	3
pinctrl	pinctrl-axp209.c	366	DoS	S	A	<1
power	gcharger-manager.c	2006	DoS	U	A	7
rapidio	rio_cm.c	2147	DoS	S	A	2
scsi	gcxgb4i.c	619	DoS	S	S	8
scsi	gql4_os.c	3206	DoS	S	A	7
scsi	gufs-hisi.c	546	DoS	U	A	<1
spi	spi-s3c64xx.c	294	DoS	U	S	5
spi	spi-topcliff-pch.c	1304	DoS	S	A	8
spi	spi-topcliff-pch.c	1307	DoS	S	A	8
staging	gaudio_manager.c	47	system crash	P	A	3
staging	grtw_xmit.c	1514	DoS	S	A	4
staging	grtl_phydm.c	182	system crash	S	A	1
thunderbolt	property.c	177	DoS	S	A	1
thunderbolt	property.c	550	DoS	S	A	1
tty	gmain.c	115	DoS	S	A	8
tty	gmain.c	135	DoS	U	A	8
tty	g8250_lpss.c	175	DoS	U	C	2
tty	gatmel_serial.c	1285	DoS	U	A	5
tty	gmxs-auart.c	1688	DoS	S	S	8
usb	gu132-hcd.c	3203	DoS	U	C	11
usb	galauda.c	438	DoS	U	S	13
usb	galauda.c	439	DoS	U	S	13
video	ghgafb.c	287	DoS	S	A	14
video	gimsttfb.c	1517	DoS	S	A	13
video	gomapdss-boot-init.c	113	DoS	U	A	3
affs	file.c	940	DoS	S	C	14
btrfs	extent-tree.c	7042	reliability	S	A	2
ipv6	gip6t_srh.c	212	DoS	S	A	1
ipv6	gip6t_srh.c	225	DoS	S	A	1
ipv6	gip6t_srh.c	235	DoS	S	A	1
openvswitch	datapath.c	449	DoS	U	A	7
smc	smc_ism.c	290	system crash	S	S	<1
strparser	strparser.c	552	DoS	S	A	2

Table 4: List of new bugs (1-142) detected with CRIX. LP = Latent Period of bugs in years. Column `Category` specifies the category of peer-slice set used to identify the bugs. A, P, S, and U indicate categories `Source-Arg`, `Source-Param`, `Source-Ret`, and `Use-Param` respectively. The `S`,`C`,`A` in the `Status` field represent patch status, `Submitted`, `Confirmed`, `Applied`, respectively.

Subsystem	Filename	Line#	Impact	Category	Status	LP
security	inode.c	339	reliability	S	A	5
ceph	osdmap.c	1900	DoS	S	S	7
isa	gsb8.c	113	reliability	U	A	14
pci	gechoaudio.c	1956	DoS	U	A	12
soc	gcs43130.c	2324	DoS	S	A	1
soc	grt5645.c	3452	system crash	U	A	<1
soc	soc-pcm.c	1236	system crash	S	S	4
md	raid10.c	3958	system crash	S	A	7
md	raid5.c	7399	system crash	S	A	7
usb	gusb_stream.c	106	DoS	S	A	10
usb	gusb_stream.c	107	DoS	S	A	10
ata	sata_dwc_460ex.c	1055	DoS	U	S	4
block	nbd.c	2117	DoS	U	S	2
net	gbcmmii.c	217	DoS	U	S	<1
slimbus	qcom-ngd-ctrl.c	1351	reliability	U	A	<1
ncsi	ncsi-netlink.c	253	reliability	U	A	1
ncsi	ncsi-netlink.c	257	DoS	U	A	1
openvswitch	conntrack.c	2146	DoS	U	S	1
openvswitch	datapath.c	466	DoS	U	A	4
openvswitch	datapath.c	475	DoS	U	A	4
openvswitch	datapath.c	477	reliability	U	A	4
tipc	group.c	942	DoS	U	A	<1
tipc	group.c	946	system crash	U	A	<1
tipc	socket.c	3226	DoS	U	A	4
tipc	socket.c	3231	reliability	U	A	4
extcon	extcon-axp288.c	145	reliability	S	A	4
thunderbolt	switch.c	1325	DoS	S	S	2
thunderbolt	xdomain.c	540	DoS	S	A	1
usb	gusb251xb.c	600	DoS	U	A	2
tty	gmax310x.c	1421	DoS	U	A	5
tty	gmvebu-uart.c	791	DoS	S	S	1
mtd	gvf610_nfc.c	856	DoS	S	A	3
mfd	mc13xxx-i2c.c	82	DoS	U	S	6
pinctrl	gberlin-bg4ct.c	453	DoS	U	S	3
pinctrl	gpinctrl-as370.c	334	DoS	U	S	<1
mfd	mc13xxx-spi.c	160	DoS	S	S	6
firmware	gdriver.c	801	DoS	S	A	2
net	gtls.c	227	DoS	U	A	<1
mmc	gdw_mmc-exynos.c	556	DoS	U	S	6
mmc	gdw_mmc-k3.c	461	DoS	S	S	5
mmc	gdw_mmc-pltfm.c	84	DoS	S	S	5
pci	gpci-host-generic.c	85	DoS	U	S	3
scsi	gtc-dwc-g210-pltfrm.c	63	DoS	U	S	3
soc	gsirf-audio-codec.c	466	system crash	S	A	5
slimbus	qcom-ngd-ctrl.c	1333	DoS	S	S	<1
x86	ghpet.c	79	DoS	U	A	11
udf	super.c	575	system crash	S	S	1
nfc	llcp_sock.c	726	DoS	S	A	7
scsi	gufshcd.c	1759	DoS	S	S	<1
thunderbolt	xdomain.c	771	DoS	S	A	1
scsi	gufshcd.c	1786	DoS	S	S	1
thunderbolt	icm.c	475	DoS	U	A	1
fmc	fmc-fakedev.c	283	DoS	S	S	5
usb	gsierra_ms.c	197	system crash	S	A	2
staging	gvchiq_2835_arm.c	212	DoS	S	C	4
thunderbolt	property.c	581	DoS	S	A	3
thunderbolt	property.c	582	buffer overflow	U	A	1
x86	gtlb_uv.c	2144	DoS	S	A	2
x86	gtlb_uv.c	2147	DoS	S	A	4
nfc	gse.c	329	DoS	U	S	1
gpio	gpio-aspeed.c	1227	DoS	S	A	2
soc	grt5663.c	3472	buffer overflow	S	C	2
soc	grt5663.c	3513	DoS	U	C	1
gpu	gv3d_drv.c	103	system crash	S	A	3
net	gmcr20a.c	534	system crash	S	A	5
net	gmcr20a.c	541	reliability	S	S	3
net	gmcr20a.c	546	reliability	S	S	3
media	gtda18250.c	705	reliability	S	C	2
soc	gcs35l34.c	263	reliability	S	S	7
dma	gomap-dma.c	1056	system crash	A	S	6
firmware	edd.c	279	system crash	A	S	4
net	gcfg80211.c	2302	system crash	A	S	4
rtc	rtc-ds1374.c	449	reliability	S	S	2
rtc	rtc-rx8010.c	193	system crash	S	S	2
mfd	tps65010.c	431	DoS	U	S	2
net	grx.c	732	DoS	U	C	<1
net	grx.c	733	DoS	U	S	3
usb	grealtek_cr.c	815	reliability	S	S	5
net	glag_conf.c	307	DoS	U	S	6
net	gmesh.c	799	system crash	S	S	2
net	gmesh.c	800	system crash	S	S	2
net	lag_conf.c	307	DoS	U	S	<1
net	p2p.c	1527	concurrency	S	S	6
message	mptctl.c	406	concurrency	S	S	10
message	mptscsih.c	1617	concurrency	S	S	10
message	mptsas.c	4803	concurrency	S	S	10
misc	tifm_7xx1.c	280	concurrency	S	S	4
pci	pcie-designware-host.c	309	DoS	U	S	1
gpu	virtgpu_kms.c	62	DoS	U	S	4
gpu	virtgpu_vq.c	48	DoS	U	S	6
input	usbtouchscreen.c	1076	DoS	U	S	8
usb	iuu_phoenix.c	369	DoS	U	S	12
usb	iuu_phoenix.c	177	DoS	U	S	12
usb	iuu_phoenix.c	729	DoS	U	S	12
usb	iuu_phoenix.c	389	DoS	U	S	12
usb	iuu_phoenix.c	253	DoS	U	S	12
usb	kobil_sct.c	248	DoS	U	S	4
usb	kobil_sct.c	339	DoS	U	S	4
usb	kobil_sct.c	354	DoS	U	S	4
usb	kobil_sct.c	284	DoS	U	S	3
ncsi	ncsi-netlink.c	250	DoS	U	S	3
openvswitch	conntrack.c	2131	reliability	S	S	<1
media	cx231xx-input.c	91	DoS	U	S	4
net	testmode.c	242	DoS	U	S	8
dma	fsl-edma-common.c	540	reliability	S	S	<1
dma	coh901318_lli.c	41	reliability	S	S	10
mtd	generic.c	69	reliability	S	S	3
net	e1000_hw.c	1046	buffer overflow	S	S	5
mfd	vx855.c	104	reliability	S	S	8
mfd	ab3100-core.c	926	reliability	S	S	8
crypto	cryptd.c	745	reliability	S	S	1
hwmon	ad7418.c	90	buffer overflow	S	S	3
hwmon	lm92.c	135	buffer overflow	S	S	3
scsi	gdth.c	5203	buffer overflow	S	S	4
staging	mmal-vchiq.c	1847	DoS	U	S	1
fsi	fsi-core.c	1250	DoS	U	S	2
net	cxgb3_offload.c	1268	DoS	U	S	7
iio	mxs-lradc-adc.c	470	DoS	U	S	1
net	myri10ge.c	2287	reliability	S	S	11
gpu	si.c	3614	reliability	S	S	6
slimbus	qcom-ngd-ctrl.c	1343	DoS	U	S	<1
net	e1000_hw.c	141	reliability	S	S	10
net	e1000_hw.c	1043	reliability	S	S	10
mtd	bcm63xxpart.c	65	buffer overflow	S	S	3
gpu	vc4_plane.c	1011	reliability	S	S	1
ext4	super.c	5866	reliability	S	S	8
net	event.c	105	buffer overflow	S	S	3
net	pch_gbe_main.c	1476	DoS	U	S	8
net	isl_ioctl.c	190	reliability	S	S	13
gpu	ast_mode.c	1201	reliability	S	S	7
hid	wacom_sys.c	2351	reliability	S	S	5
media	ov9650.c	609	buffer overflow	S	S	<1
soc	sti_uniperif.c	292	reliability	S	S	2
media	em28xx-cards.c	3987	reliability	S	S	2
usb	xhci-pci.c	269	reliability	S	S	1
net	nic_main.c	1229	crash	S	S	3

Table 5: Continued list of new bugs (143-278) detected with CRIX. LP = Latent Period of bugs in years. Column `Category` specifies the category of peer-slice set used to identify the bugs. A, P, S, and U indicate categories `Source-Arg`, `Source-Param`, `Source-Ret`, and `Use-Param` respectively. The `S,C,A` in the `Status` field represent patch status, `Submitted`, `Confirmed`, `Applied`, respectively.