2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W) | 978-1-6654-3950-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/DSN-W52860.2021.00028

Automatically Constructing Peer Slices via Semantic- and Context-Aware Security Checks in the Linux Kernel

Yongzhi Liu
School of Software and Microelectronics
Peking University
Beijing, China
lyz_cs@pku.edu.cn

Xiarun Chen
School of Software and Microelectronics
Peking University
Beijing, China
xiar_c@pku.edu.cn

Zhou Yang
School of Software and Microelectronics
Peking University
Beijing, China
yzss2019@pku.edu.cn

Weiping Wen
School of Software and Microelectronics
Peking University
Beijing, China
weipingwen@ss.pku.edu.cn

***Abstract*—OS kernels enforce many security checks to validate system states. We observe that paths containing security checks are in fact very informative in inferring critical semantics in OS kernel. In particular, Such slices are valuable for detecting kernel semantic bugs because understanding semantics is typically required by the detection. However, there are few studies that address security checks, and constructing these slices is challenging due to not only a lack of clear criteria but also the large and complex OS. In this paper, combining security checks with program slicing, we first systematically study security check peer slices and propose an automatic approach to construct security check peer slices in OS kernel. Using an inter-procedural, semantic- and context-aware analysis, we can find slices sharing similar semantics in similar contexts. Based on the information offered by security check peer slices, we then introduce the Scenarios for semantic vulnerability detection by security check peer slices: missing security check and inaccurate security check. The evaluation results show that our approach can accurately constructing security check peer slices.**

Keywords—program slicing, security check, static analysis, OS kernel, semantic bug

I. INTRODUCTION

Operating systems are computer programs that manage computer hardware and software resources and are complex and large in size, with more than 25 000 000 lines of code in the Linux kernel alone. Due to its complex logic, operating systems are often subject to errors and other anomalies, such as unexpected behavior from software, abnormal input from the operator, physical failure of hardware, etc. Security check is the most common security enhancement in operating systems, which checks the critical variables to confirm the current operating state of the system to ensure the system's security. The existing security checking mechanism does not perfectly solve the problems in the system, and missing[1, 2, 3] or incorrect [4]security checking has been a very common situation in operating systems, which can lead to malicious code attacks (e.g., stack overflow, etc.) or fatal errors (e.g., system crash, etc.) in the process of operation[5].

Unfortunately, even though the problem of lack of security checks is so widespread and serious, there are few research projects on security checks because of several inherent challenges. (1) System vulnerabilities regarding security checks have not received much attention. (2) Operating system source code is so large and complex that detailed analysis of the source code is an impossible task. (3) There are also few studies for identifying exception handling and error codes. To address the above three points, we conduct an in-depth study on peer semantic slicing of Linux OS security checks, so as to achieve accurate detection of kernel semantic vulnerabilities through security check slices.

System software has a large number of variables, and important variables (critical variables) are often protected by security checks. It is worth noting that critical variables are also passed between procedures, so code snippets with the same semantics also need to have the corresponding security checks added, which are often forgotten by developers, leadingto the missing security check bug. We refer to these program slices with similar semantics and context as the security check peer slices. For example, line 33 in Figure 1 indicates that the function pointer variable *write* can make indirect calls to functions whose call targets change as the control flow changes, thus generating multiple peer semantic paths, one of which is *coalesced_mmio_write* with two security checks. Further, we can speculate that the *coalesced_mmio_write* function contains four security-checked slices. If peer paths exist, then the slices for security checks on the same variable are peer. Our source code analysis reveals that the variable *last* can be controlled by the user and thus lines 22-25 can lead to potentially arbitrary write. To avoid out-of-bounds access, "within-boundary" should be checked for security before the variable *last* is used. This information is valuable for detecting semantic vulnerabilities. By constructing security check peer slices, we can inspect whether a critical variable has been properly security checked before it is used, and also determine whether the security checks that already exist are accurate.

It is worth noting that it is useful to construct security check peer slices for fuzzing and system hardening. Fuzzing techniques[6] are often inefficient in exploring deep paths due to traversing invalid paths[7, 8]. On the other hand, existing system hardening techniques such as memory security[9] and

978-1-6654-3950-3/21/$31.00 ©2021 IEEE
DOI 10.1109/DSN-W52860.2021.00028

fault/memory isolation[10, 11]tend to require very high performance expenses. Constructing semantic slices of security check peers allows researchers to selectively focus on a few targets that need to be protected, thus improving performance.

```
1  //Linux 5.3: /virt/kvm/coalesced_mmio.c
2  static int coalesced_mmio_write(struct kvm_vcpu *vcpu,   ②
3                                  struct kvm_io_device *this, gpa_t addr,
4                                  int len, const void *val)
5  {
6          struct kvm_coalesced_mmio_dev *dev = to_mmio(this);
7          struct kvm_coalesced_mmio_ring *ring = dev->kvm->coalesced_mmio_ring;
8          // The end identifier of slice1, slice2 and slice3.
9          if (!coalesced_mmio_in_range(dev, addr, len))
10                 return -EOPNOTSUPP;
11
12         spin_lock(&dev->kvm->ring_lock);
13
14         // The end identifier of slice4.
15         if (!coalesced_mmio_has_room(dev)) {
16                 spin_unlock(&dev->kvm->ring_lock);
17                 return -EOPNOTSUPP;
18         }
19
20         /* copy data in first free entry of the ring */
21         // ERROR: out-of-bounds access.
22         ring->coalesced_mmio[ring->last].phys_addr = addr;
23         ring->coalesced_mmio[ring->last].len = len;
24         memcpy(ring->coalesced_mmio[ring->last].data, val, len);
25         ring->coalesced_mmio[ring->last].pio = dev->zone.pio;
26         smp_wmb();
27         ring->last = (ring->last + 1) % KVM_COALESCED_MMIO_MAX;
28         spin_unlock(&dev->kvm->ring_lock);
29         return 0;
30 }
31 static const struct kvm_io_device_ops coalesced_mmio_ops = {
32         // Initialization of the head identifier.
33         .write      = coalesced_mmio_write,
34         .destructor = coalesced_mmio_destructor,
35 };  ①
```

Figure. 1. Four security check slices (Line 9 and 15). One slice is used for *addr*, one for *len* and two for *dev*.

Although building security check peer slices shows great opportunities in detecting dangerous semantic bugs and system hardening, etc., it is also a very challenging task. First, identifying security checks requires semantic understanding. Not all conditional statements are security checks, and only a small fraction of them are satisfied. In order to improve the accuracy of semantic program slices, false positives and false negatives of identifying security checks need to be reduced. In addition, in terms of code slicing, existing slicing methods are code decomposition-based in nature, while bug discovery-oriented slicing wants to reflect the context of bugs from input to defect triggering. therefore, suitable slicing models and algorithms expressing vulnerability need to be established. Finally, the operating system kernel is extremely large and complex. Program semantic analysis of it is very challenging, and corner cases such as hand-written assembly make the analysis results error-prone.

In this paper, we aim to automate the construction of security check peer slices in complex and large operating system kernels, and introduce scenarios where these slices are used to detect some common and critical kernel semantic bugs. In order to address the previously mentioned challenges and effectively detect kernel semantic vulnerabilities, we first complete a study of security-checking peer slicing to better understand its properties. Based on this study, we further introduce the application scenarios of security check peer slices in vulnerability detection.

We make some contributions in this paper as follows.

- **A study of security check peer slices.** We conduct a study of security check peer slices to find intrinsic differences between security check peer slices and normal code paths, and to discover the semantic relationships between security check slices.

- **Automatic construction of security check peer slices.** We propose an automatic approach to construct security check peer slices in OS kernel. The approach incorporates multiple techniques such as identifying extensively security checks and identifying targets of indirect call.

- **Scenarios for semantic vulnerability detection.** We present scenarios of security check peer slices for vulnerability detection: missing security check bugs, inaccurate security check bugs. In addition, we then present scenarios of its application in code auditing.

II. BACKGROUND

A. Serious Security Impact

Security checks are a class of conditional statements that can be used to verify the execution status of a program. As we know, the detection of security checks is of great significance for vulnerability discovery and vulnerability exploitation, and the lack of security checks or incomplete security checks in operating systems may cause very serious consequences, even leading to system crashes. As the most important indicator for detecting security check vulnerabilities, the security check slice is also crucial to system security.

In order to investigate the significant impact of security checks, we conducted additional research based on the literature [3] in the following areas: (1) How many security bugs are caused by the lack of security checks, (2) The serious security impact of these security vulnerabilities.

In response to the first question, we surveyed 300 randomly selected security vulnerabilities from 2017 to 2019 in the U.S. *NVD*[12]. Among these vulnerabilities, we selected vulnerabilities that were fixed by adding security checks as missing security check vulnerabilities. The statistical results show that a total of 184 vulnerabilities are about security check vulnerabilities. In 2017 and 2018, the majority (59.5%) of the security vulnerabilities were related to missing security checks, while in 2019, the percentage was as high as 65%.

To address the second issue, we learned from our survey that 11 of the most serious vulnerabilities from 2017 to 2019, i.e., those with a *CVSS* score of 10 (the highest severity level), were caused by a lack of security checks. According to our statistics, in 2019, among the 65 security check vulnerabilities obtained from our sample, 7 vulnerabilities were identified as Critical and 28 as High in *CVSS*, which means that about 53.9% of the missing security check vulnerabilities are more serious.

B. Lack of Relevant Research

Little research has been done on security check slices. To the best of our knowledge, this area of research is mainly represented in Kangjie Lu's CRIX[3], which improves the error code identification method mentioned in LRSan[13] by using both error codes and exception handling functions as the benchmark for identification, thus identifying more security checks and critical variables, and finally performing forward and backward data flow analysis on the critical variables to find all the security check slices. However, we find that the slices

mentioned in CRIX are all terminated by conditional statements or path endings, which causes a large number of false positives and inaccurate semantic identification. For example, as shown in Figure 2, CRIX does not include check functions in the end identifiers and thinks *hcd_pci_suspend_noirq* does not do a security check on *dev*, but *check_root_hub_suspended* called in *hcd_pci_suspend_noirq* have done it. It is thus clear that slices that end only with conditional statements or path endings are imperfect. Even though this approach suffers from certain false positives and false negatives, it still provides very much support for security check-based vulnerability detection.

As the cornerstone of security check-based vulnerability detection, the more accurate the security check slices are identified, the better their detection results will be, and the lack of existing work, in terms of for slice construction, is the biggest motivation for our research. In III, we give a definition of the security check peer slice.

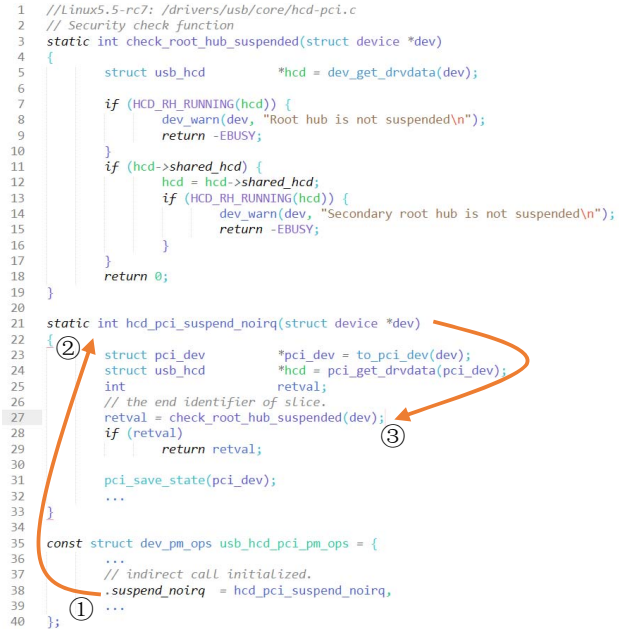

Figure. 2. An example of a security check function

C. Slice Conception

This section describes some terms and concepts about program slice. The process of removing irrelevant statements from a program and thus reducing the complexity of program analysis is called program slicing. In other words, a slice is a miniature version of a source code program. According to Weiser, a static slice is a collection of statements that can directly or indirectly affect the values of variables in a given program point, which is called a slicing criterion[14]. This slicing criterion is denoted as *(S,V)*, where *S* is the statement or line number and *V* is the variable in the program. The feedback using program slicing in many applications is not satisfactory. This is because in many scenarios the size of the slices obtained is not significantly smaller than the original program. For large programs, a single slice may contain dozens or even hundreds of program instructions and miss program semantics. Similarly, in security check scenarios, the size of security check slices obtained can also be large, as well as the association between instructions in the slices is not clear. To eliminate this drawback, we introduce security check peer slices to reduce the size of slices and improve program semantics.

III. A STUDY OF SECURITY CHECK PEER SLICES

There are a large number of security check statements in the operating system kernel. Before a critical variable is used, the system often checks it for security to ensure system stability. Due to the call relationship between functions, there are many paths with similar semantics and contexts. In the *coalesced_mmio_write* function shown in Figure 1, different arguments and different parameter fields can lead to different paths. The security check at line 15 corresponds to the example where the argument *this* is the same as its parameter field, as it performs a security check on *this*. In order to better understand the security check peer slices, we need to give a definition of security check peer slices. For this purpose, we have investigated some ground-truth security checks. There are many OS bugs that are fixed by inserting a security check, and the check is often on the variable being used or its fields. We have collected 40 security checks from previous papers[2, 15]. In addition, 50 were collected from the Linux kernel's git patch history. Based on the intrinsic features of the security checks, we provide the definition of security check peer slices.

A. Definition of Security Check Peer Slice

We first give a definition of security check peer slice based on the intrinsic features of security check[16] and our above investigation. We know that security check is adding conditional constraints to critical variables, so the target of the security check peer slice is critical variables. Let the critical variable be *CV*. $\varphi(CV)$ denotes the field set of *CV*. $\varphi(CV)_i$, where *i* is an integer index, represents the "*i*th" field of *CV*. The head of the slice is *S*, and *E* denotes the end of the slice, so then the security check slice denotes the set of statements starting with *S* and ending with *E*. We then identify the statement set as a security check slice if the statement where *S* or *E* is located belongs to the set which includes security check conditional statement *SC*, security check function *SCF*, and other statement containing *CV*, where *SC* or *SCF* makes security check on *CV* or $\varphi(CV)_i$.

If two such security check slices have the same semantics and context, then we call them peer semantic and peer context. In other words, the construction of security check peer slices relies heavily on the selection of slicing criterion. An effective and practical slicing criterion can avoid the path explosion problem[17] and yet ensure that our slices are peer-to-peer. Therefore, a good slicing criterion is crucial to reduce the false positive rates, which will be described in the next section.

B. Slicing Criterions

In this section, we study some slicing criterions for generating peer paths. seemingly, we can perform forward slicing and backward slicing for each critical variable to find all the security check peer slices. Such a naive approach is prone to high time overhead and high false positive rates. To solve these problems, we must construct peer slices for the source and use

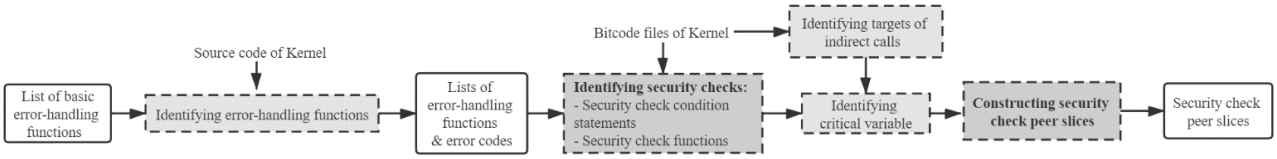

Figure. 3. The workflow of SCSlicer. SCSlicer takes as input the kernel source code and some basic exception handling functions to automate the construction of security check peer slices

of critical variables. For program control flow graphs, call instructions and return instructions tend to generate peer paths. Therefore, we summarize three semantic slicing criterions.

Criterion-1(n_1, CV): *CV* represents the critical variable required for slicing, and n_1 denotes the program point where the function with CV as an argument is indirectly called. As shown in the example in Figure 4, the indirect call *dev->ops-> write()* is equivalent to a dispatcher, which is called against functions that share similar semantics, such as *coalesced_mmio_write* in Figure 1 and *ioeventfd_write* in *eventfd.c*[18]. These callees take arguments from the same caller, so they also have similar contexts.

```
1  //Linux5.3: /include/kvm/iodev.h
2  static inline int kvm_iodevice_write(struct kvm_vcpu *vcpu,
3                                       struct kvm_io_device *dev, gpa_t addr,
4                                       int l, const void *v)
5  {
6          /* Indirect call: "dev->ops" is of type "kvm_io_device_ops" */
7          return dev->ops->write ? dev->ops->write(vcpu, dev, addr, l, v)
8                                 : -EOPNOTSUPP;
9  }
10
```

Figure. 4. An example of a function pointer as a field of a struct

Criterion-2(n_2, CV): *CV* also represents the critical variable required for slicing, and n_2 represents the program point where the function is called with *CV* as the return value or output argument. If a function *callee* returns a critical variable or uses a critical variable to assign an output argument, then the functions calling the *callee* share similar semantics. The return values of the callers all come from the same callee, so they also share similar contexts.

Criterion-3(n_3, CV): *CV* represents the critical variable required for slicing, and n_3 represents the program point where the function with *CV* as the parameter is called. When the critical variable comes from an argument to the current function *callee*, the functions calling the *callee* share similar semantics. The arguments of callers are all passed to the same callee, then they also are used as similar semantics in similar contexts.

IV. Experimental Evaluation

We implement an experimental tool, *SCSlicer*[19], using the above research. Then, we collect some results related to security check slices to extensively evaluate the scalability and effectiveness of *SCSlicer* using the Linux kernel.

A. SCSlicer: Security Check Peer Slicer

Based on the study in III, we implement a security check peer slicing tool, *SCSlicer*. Figure 3 illustrates the workflow of SCSlicer. Since the identification of security checks relies on precise exception handling functions, the first step of *SCSlicer* is to identify a more comprehensive set of exception handling functions. Based on exception handling functions and error codes, *SCSlicer* extends the security check identification algorithm mentioned in [14] to generate sets of security check conditional statements and security check functions. With the sets, our second step is to identify indirect call targets to construct an accurate call graph, thus identifying the source and use of critical variables. We find the target of an indirect call by matching the number and type of arguments of a function pointer, for which an assignment relationship exists, with the target function. Finally, we construct security check peer slices using the three slicing criterions described in III.B..

B. Evaluating the Constructing of Security Check Peer Slices

In this section, we extensively evaluate *SCSlicer* using Linux Kernel of version 5.5-rc7 with the top git commit number def9d2780727cec. The experimental environment is conducted on an Ubuntu 16.04 LTS system with LLVM 10.0. We use *wllvm*[20] to generate a 452M LLVM IR bytecode file to cover more modules.

Statistical results. Before presenting the evaluation results, we first show some interesting statistical results, as shown in TABLE 1. With the *cloc*[21] tool, the evaluation experiment covers 18.8 million lines of Linux kernel code. For Linux, *SCSlicer* identified 46.1K security checks, which are security check conditional statements (97.3%) and security check functions (2.7%). *SCSlicer* identified 172.9K security check slices using the three slicing criteria defined above, and a total of 9.5K security check peer semantic slices. The 822 security check peer slices are identified by indirect call scenarios. *SCSlicer* found most security check peer slices (65.5%) in SC-3 scenario. Since we added the security check functions to security checks, *SCSlicer* builds more security-checking peer slices than *CRIX*. *SCSlicer* finds more security check slices and security check peer slices. If we use *SCSlicer* to audit the Linux kernel, we can find more kernel semantic bugs.

False negatives. *SCSlicer* constructed security check peer slices via the three slicing criterions. *SCSlicer* may have false negatives if not all slicing criterions are covered. However, other slicing criterions are difficult to solve program semantic problems because they are not easily semantic- and context-aware.

False positives. *SCSlicer* found 9.5K security check peer slices, however *CRIX* identified 5.4K. To evaluate false positive, we randomly selected 500 slices reported by the two tools for Linux kernel, and confirmed whether they are real security check peer slices based on the definition described in III. The results show that 467 of them are true positives.

TABLE 1: SOME STATISTICAL NUMBERS RELATED TO SECURITY CHECK SLICES

Tool	Security checks		Security check slices	Security check peer slices	Through SC-1	Through SC-2	Through SC-3
	Conditional statements	Functions					
CRIX	42.6K	0	116.2K	5.4K	709	1463	3254
SCSlicer	44.9K	1.2K	172.9K	9.5K	822	2464	6235

Scalability. The experiments were performed on an Ubuntu 16.04 LTS system with LLVM version 10.0 installed. The machine has a 8 GB RAM and an Intel CPU (Core™ i5-8265U 1.6GHz) with 8 cores. *SCSlicer* is faster than *CRIX*—it finished the whole analysis for Linux within 10 minutes: three minutes are for loading bitcode files and identifying security checks, and seven minutes is for constructing security check peer slices.

V. APPLICATION OF SECURITY CHECK PEER SLICES

Security check peer slices are very informative in revealing critical semantics. With them, a bug related to security check is more likely to be detected or triggered. Without such information, a bug detected by previous approaches[1, 2, 3, 4, 16, 22, 23] may not be critical at all or is a false positive. Also, we may find other unchecked variables on the security check slices.

In this section, we will present how to use these slices to effectively detect critical kernel semantic bugs: missing security check bugs and inaccurate security check bugs. In addition, we combine security check peer slices and code auditing technology to help auditors find common vulnerabilities in source code more quickly.

A. Detecting Critical Kernel Semantic Bugs

In this section, we will present how to use security check peer slices to effectively detect critical kernel semantic bugs involving missing security checks and inaccurate security checks.

Missing security checks. Missing security checks is a class of semantic bugs in software programs where erroneous execution states are not validated. We construct three different peer slices by using the targets of security check statements as a starting point and finding their source and use through taint forward and backward analysis. Using these peer semantic slices, we can detect missing security check bugs by implementing the following steps: (1) identifying security checks in peer semantic slices and modeling constraints on critical variables; (2) finding peer slices that do not do security checks on critical variables; and (3) calculating the proportion of slices with missing security check statements among all peer semantic slices to generate bugs detection reports.

Inaccurate security checks. There are a large number of security checks in the Linux kernel source code, but the accuracy of the security checks is often disputed. Inaccurate security checks can lead to serious security impacts. Inaccurate security check bugs focus on correcting security checks to prevent vulnerabilities such as some buffer overflows, as opposed to security check missing bugs. Similar to detecting missing security checks, we identify all security checks in peer semantic slices containing security checks and model constraints on critical variables. Then, we also calculate the proportion of constraints in a class of security check slices containing security check for the same critical variable. Finally, we believe that the constraint that appears most often is the correct security check.

VI. DISCUSSION

In this section, we discuss limitations of *SCSlicer* that can be potentially improved and explored as future work directions.

Slicing criterions. There are a wide variety of security check slices within the kernel. According to call instructions and return instructions tending to generate peer paths, *SCSlicer* summarizes three slicing criterions. However, in addition to criterion 1-3, whether there are other criterions is also a question worthy of discussion. An improved model can include as many slicing criterions as possible, to make the *SCSlicer* more complete.

Bugs Detection. Critical kernel semantic bugs are detected by cross-checking security check slices. We can set the threshold to a proper value to minimize the false positives at the cost of completeness. The specific vulnerabilities found through the use of this tool are our next step to do. Similarly, the method proposed in this paper can be used to detect bugs in other software other than the kernel. However, it may have lots of false positives for smaller target programs because they do not have enough security check peer slices.

VII. RELATED WORK

To the best of our knowledge, *SCSlicer* is the first systematically to construct security check peer slices. We identify two research lines that are related to *SCSlicer*: error handling analysis and missing-check detection.

Error handling analysis. Several efforts have attempted to analysis error handling. EIO[24] presents an approach that uses data-flow analysis to detect unchecked errors in the file system code. Cheq[16] locates security checks and error handling functions in the kernel by searching certain patterns and uses this information to detect bugs. EPEx[25] and APEx[26] identify code paths in a callee function that may return error codes and check if the error codes are handled in callers. Hector[27] targets the properties of error-handling code to detect resource-release bugs.

Missing-check detection. LRSan[13] detects lacking-recheck bugs, a subclass of missing-check bugs. It however uses only standard error codes without considering custom error codes and error-handling functions. With our tool *SCSlicer*, We can detect general missing-check bugs and inaccurate-check bugs that include lacking-recheck bugs. Crix[3] cross-checks a property and uses statistical analysis to detect missing-check

bugs, but constructs less peer slices compared with *SCSlicer* for security check functions are not considered; It cannot detect inaccurate-check bugs because the cross-checking target is all peer slices. Utilizing cross-checking between existing implementations of file systems, Juxta[2] detect semantic bugs such as missing-check bugs. *SCSlicer* can analyze all subsystems in the Linux kernel. There are also a few other complementary approaches to detect missing-check. Chucky[1] uses check deviations to infer missing check. MACE[28] is an annotation based on static analysis framework and it can find missing authorization checks in web applications. Pex [4] uses implicit programming rules to find missing, inconsistent, and redundant permission checks.

VIII. CONCLUSION

We presented our study on security check peer slices and designed an automated experimental tool, *SCSlicer*, for precisely constructing security check peer slices in OS kernel. The Construction is semantic- and context-aware with an inter-procedural data flow analysis. In addition, to find common vulnerabilities more quickly, we introduce the Scenarios for semantic vulnerability detection by security check peer slices: missing security check and inaccurate security check. We believe that construction of security check peer slices could facilitate future research on the semantic bug detection.

REFERENCES

[1] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: exposing missing checks in source code for vulnerability discovery," 2013, pp. 499-510.

[2] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: the case of finding file system bugs," 2015, pp. 361-377.

[3] K. Lu, A. Pakki, and Q. Wu, "Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences," 2019, pp. 1769-1786.

[4] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "PeX: A Permission Check Analysis Framework for Linux Kernel," 2019, pp. 1205-1220.

[5] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang, "Characterization of Linux Kernel Behavior under Errors," 2003, pp. 459-468.

[6] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," 2018, pp. 2123-2138.

[7] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, T. Kim, and U. Assoc, "CAB-FUZZ: Practical Concolic Testing Techniques for COTS Operating Systems," *2017 Usenix Annual Technical Conference*, pp. 689-701, 2017.

[8] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, Z. Chen, and Ieee, "CollAFL: Path Sensitive Fuzzing," *2018 Ieee Symposium on Security and Privacy*, IEEE Symposium on Security and Privacy, pp. 679-696, 2018.

[9] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," *Acm Sigplan Notices,* vol. 44, no. 6, pp. 245-258, Jun, 2009.

[10] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," 2017, pp. 437-452.

[11] L. Mogosanu, A. Rane, and N. Dautenhahn, "MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation," 2018, pp. 359-379.

[12] " NVD NIST Search," https://nvd.nist.gov/.

[13] W. Wang, K. Lu, and P.-C. Yew, "Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels," 2018, pp. 1899-1913.

[14] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.,* vol. 10, no. 4, pp. 352-357, /, 1984.

[15] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel," 2017, pp. 1-16.

[16] K. Lu, A. Pakki, and Q. Wu, "Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs," 2019, pp. 3-25.

[17] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "Path-Sensitive Backward Slicing," 2012, pp. 231-247.

[18] "eventfd.c," https://github.com/torvalds/linux/blob/v5.3/virt/kvm/eventfd.c.

[19] "SCSlicer," https://github.com/LinusRobot/SCSlicer.

[20] "wllvm," https://pypi.org/project/wllvm/.

[21] "CLOC," http://cloc.sourceforge.net/.

[22] I. Dillig, T. Dillig, and A. Aiken, "Static error detection using semantic inconsistency inference," 2007, pp. 435-445.

[23] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. R. Engler, "From Uncertainty to Belief: Inferring the Specification Within," 2006, pp. 161-176.

[24] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: Error Handling is Occasionally Correct," 2008, pp. 207-222.

[25] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically Detecting Error Handling Bugs Using Error Specifications," 2016, pp. 345-362.

[26] Y. J. Kang, B. Ray, and S. Jana, "APEx: automated inference of error specifications for C APIs," 2016, pp. 472-482.

[27] 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," 2013, pp. 1-12.

[28] M. Monshizadeh, P. Naldurg, and V. N. Venkatakrishnan, "MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications," 2014, pp. 690-701.