Static Detection of Unsafe DMA Accesses in Device Drivers

Jia-Ju Bai
Tsinghua University

Tuo Li
Tsinghua University

Kangjie Lu
University of Minnesota

Shi-Min Hu
Tsinghua University

Abstract

Direct Memory Access (DMA) is a popular mechanism for improving hardware I/O performance, and it has been widely used by many existing device drivers. However, DMA accesses can be unsafe, from two aspects. First, without proper synchronization of DMA buffers with hardware registers and CPU cache, the buffer data stored in CPU cache and hardware registers can be inconsistent, which can cause unexpected hardware behaviors. Second, a malfunctioning or untrusted hardware device can write bad data into system memory, which can trigger security bugs (such as buffer overflow and invalid-pointer access), if the driver uses the data without correct validation. To detect unsafe DMA accesses, some key challenges need to be solved. For example, because each DMA access is implemented as a regular variable access in the driver code, identifying DMA accesses is difficult.

In this paper, we propose a static-analysis approach named SADA, to automatically and accurately detect unsafe DMA accesses in device drivers. SADA consists of three basic steps. First, SADA uses a field-based alias analysis to identify DMA accesses, according to the information of DMA-buffer creation. Second, SADA uses a flow-sensitive and pattern-based analysis to check the safety of each DMA access, to detect possible unsafe DMA accesses. Finally, SADA uses an SMT solver to validate the code-path condition of each possible unsafe DMA access, to drop false positives. We have evaluated SADA on the driver code of Linux 5.6, and found 284 real unsafe DMA accesses. Among them, we highlight that 121 can trigger buffer-overflow bugs and 36 can trigger invalid-pointer accesses causing arbitrary read or write. We have reported these unsafe DMA accesses to Linux driver developers, and 105 of them have been confirmed.

1 Introduction

A modern operating system (OS) controls different kinds of peripheral hardware devices, including Ethernet controllers, sound cards, storage adapters and so on. To improve the performance of data communication between the OS and hardware devices, Direct Memory Access (DMA) is designed to reduce CPU involvement for hardware I/O. The OS enables DMA by mapping hardware registers to an area of system memory, which is called DMA buffer, and then the OS can directly access the hardware registers by accessing the DMA buffer.

Many existing device drivers have used DMA to improve performance, but DMA accesses can be unsafe, even though IOMMU has been used to guarantee their accessed memory addresses are valid. First, the driver should access the DMA buffer only when the buffer has been properly synchronized with hardware registers and CPU cache. Otherwise, the accessed data stored in hardware registers and CPU cache can be inconsistent, which can cause unexpected behaviors of the hardware device. For short, we call such a problem as *inconsistent DMA access*. Second, considering that a hardware device can be malfunctioning [27, 55] or untrusted [28, 53, 65], it can write bad data into DMA buffers, and thus the driver should perform correct validation of the data from DMA buffers before using it. Otherwise, security bugs (such as buffer overflow and invalid-pointer access) can be triggered at runtime. For short, we call such a problem as *unchecked DMA access*.

To mitigate the security risks from DMA accesses, several recent works [45, 51, 52] perform driver fuzzing and have found some security bugs caused by the bad data from DMA buffers. Specifically, they create a simulated device to generate and mutate hardware inputs (including the data from DMA buffers), and test whether the driver can correctly handle these inputs. But they still have some limitations in detecting unsafe DMA accesses. First, they require associated simulated devices to actually run the tested drivers, and implementing such simulated devices often requires much manual work. Second, their code coverage is limited to generated test cases, causing that many real unsafe DMA accesses are missed. Finally, they cannot detect inconsistent DMA accesses, because they do not consider the synchronization of DMA buffers.

Static analysis is effective in achieving high code coverage and reducing false negatives. But using static analysis to detect unsafe DMA accesses in the Linux driver code is still challenging. First, as each DMA access is implemented

as a regular variable access in the driver code, it is difficult to statically identify DMA accesses. Second, as the Linux kernel code base is very large and complex, performing static analysis of it is also difficult. Third, static analysis can report many false positives due to lacking exact runtime information of the driver. To our knowledge, there is no systematic static approach of detecting unsafe DMA accesses at present.

In this paper, we propose a static-analysis approach named SADA (Static Analysis of DMA Accesses), to automatically and accurately detect unsafe DMA accesses in device drivers. Overall, SADA consists of three basic steps. First, considering that DMA accesses and DMA mapping creation may be performed in different driver functions, SADA uses a field-based alias analysis to identify DMA accesses according to the information of DMA mapping creation, because our study of the Linux driver code finds that about 87% of created DMA buffers are stored in data structure fields in the driver code. Second, SADA uses a flow-sensitive and pattern-based analysis to check the safety of each DMA access, to detect possible unsafe DMA accesses. Specifically, to detect inconsistent DMA accesses, SADA checks whether each DMA access is performed with proper synchronization of DMA buffers by analyzing code context. To detect unchecked DMA access, SADA uses a static taint analysis to check whether the accessed data from DMA buffers can cause possible insecure influence on data flow or control flow. For example, if a variable stored in a DMA buffer is used as an array index without any check, a buffer-overflow bug can occur. Finally, SADA uses an SMT solver Z3 [66] to validate the code-path feasibility of each possible unsafe DMA access, to drop false ones. In this way, the overhead introduced by the SMT solver can be reduced compared to the traditional way of validating code-path condition while analyzing the whole driver code. We have implemented SADA with LLVM [33].

Overall, we make the following technical contributions:

- By studying DMA in device drivers, we reveal the security risks of DMA accesses from two aspects: 1) they can cause unexpected hardware behaviors; and 2) they can trigger security bugs (such as buffer overflow and invalid-pointer access) caused by the bad data from malfunctioning or untrusted hardware devices.
- We propose a practical static-analysis approach named SADA, to effectively detect unsafe DMA accesses in device drivers. SADA incorporates multiple techniques to ensure the precision and effectiveness of the detection. To our knowledge, SADA is the first systematic static approach to detect unsafe DMA accesses.
- We evaluate SADA on Linux 5.6, and find 284 real unsafe DMA accesses. Among them, we highlight that 121 can trigger buffer-overflow bugs and 36 can trigger invalid-pointer accesses causing arbitrary read or write. We have reported these unsafe DMA accesses to Linux driver developers, and 105 of them have been confirmed.

The rest of this paper is organized as follows. Section 2 introduces the background and our study of DMA. Section 3 analyzes the challenges of static detection of unsafe DMA accesses. Section 4 introduces our solution techniques. Section 5 presents SADA in detail. Section 6 shows our evaluation. Section 7 makes a discussion about SADA and unsafe DMA accesses. Section 8 introduces related work, and Section 9 concludes this paper.

2 Background and Study of DMA

In this section, we introduce DMA and its problems in existing research, then reveal the security risks of DMA accesses, and finally study DMA in Linux device drivers.

2.1 DMA Architecture

Direct Memory Access (DMA) is a popular mechanism that allows peripheral hardware devices to communicate data with system memory without CPU involvement. Without DMA, when the data is transfered between a hardware device and system memory, a CPU is typically fully occupied for the entire duration of the data transfer, and thus this CPU is unavailable to perform other tasks. With DMA, a CPU *just* initiates the data transfer and then hands over the actual data transfers to the DMA controller (DMAC) , so the CPU can focus on other tasks. Once the data transfer finishes, the CPU will receive an interrupt from the DMA controller to wrap up the data transfer. In this way, the CPU performs only the minimum jobs, namely initialization and finalization of the data transfers, which thus improves hardware I/O performance.

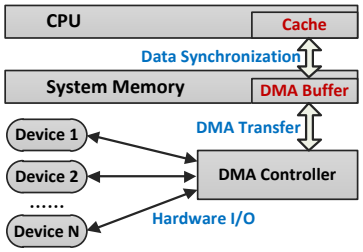

Figure 1: The DMA architecture.

Figure 1 shows the architecture of DMA in modern computer systems. To enable DMA, a DMA buffer is allocated and mapped to system memory and hardware registers. When the CPU wants to read hardware registers, it directly reads the DMA buffer in system memory and synchronizes the data into the CPU cache. Similarly, when the CPU wants to write hardware registers, it directly writes to the DMA buffer in system memory and synchronizes the data into hardware registers. To reduce programming complexity, each DMA access is implemented as a regular variable access in the driver code, such as `data = dma_buf->data` (reading a DMA buffer) and `dma_buf->data = data` (writing a DMA buffer).

Coherent DMA	
Allocate	*dma_alloc_coherent, pci_alloc_consistent, dma_pool_alloc, ...*
Release	*dma_free_coherent, pci_free_consistent, dma_pool_free, ...*
Streaming DMA	
Map	*dma_map_single, dma_map_page, pci_map_single, ...*
Unmap	*dma_unmap_single, dma_unmap_page, pci_unmap_single, ...*
Synchronize	*dma_sync_single_for_cpu, pci_dma_sync_single_for_device, ...*

Table 1: Typical DMA interfaces in the Linux kernel.

DMA type. According to the synchronization way with the hardware registers and CPU cache, there are two types of DMA buffers used in device drivers:

Coherent DMA buffer. A coherent DMA buffer is simultaneously available to both the CPU and hardware device, and it often exists for the lifetime of the driver (it is allocated in driver initialization and released in driver removal). To make the data stored in hardware registers and CPU cache always coherent, this DMA buffer must live in cache-coherent memory, which is often expensive to set up and use. In this way, the driver does not need to explicitly synchronize the data between hardware registers and CPU cache.

Streaming DMA buffer. A streaming DMA buffer is asynchronously available to both the CPU and hardware device, and it is often dynamically mapped and unmapped to a specific memory area when the driver runs. Because the data stored in hardware registers and CPU cache can be inconsistent, the driver needs to explicitly synchronize the data between them at proper time. However, because streaming DMA buffer does not live in cache-coherent memory, it is cheaper than coherent DMA buffer to set up and use.

DMA interface. The Linux kernel provides specific kernel interfaces for drivers to perform DMA operations. Table 1 lists some commonly-used interfaces for coherent and streaming DMA buffers. Note that a coherent DMA buffer is in cache-coherent memory, and thus it does not require interfaces for synchronization between hardware registers and CPU cache.

2.2 DMA Problems in Existing Research

Though DMA can improve hardware I/O performance, it also introduces security risks. In the past few years, many security problems of DMA have been found and extensively fixed, and we list representative ones as follows:

DMA attack. Through DMA, a malicious DMA-enabled hardware device can gain direct access to part or all of the system memory [21]. In this way, the attacker can steal confidential data or take control of the OS. To defend against DMA attack, many existing approaches [40–42, 44, 58] use Input-Output Memory Management Unit (IOMMU) to limit the area of system memory that a DMA-enabled hardware device can access.

Invalid mapping. A DMA buffer should be mapped to a physical memory area of contiguous addresses. For this reason, a DMA buffer cannot be mapped to stack memory, because its physical memory addresses can be non-contiguous. Otherwise, unexpected stack overflow may occur at runtime. Recently, such problems are highlighted by Linux driver developers [23], because some Linux kernel commits (such as `6c2794a2984f` [6] and `3840c5b78803` [7]) have been applied to fix such problems.

Improper checking of buffer creation. Once a DMA interface is called by the driver to create a DMA buffer, its return value should be properly checked in the driver code, because the creation can fail. Otherwise, null-pointer dereferences or invalid DMA accesses may occur. In 2013, Linux driver developers used a simple static analysis [22] to detect many such problems in the Linux kernel, and some of them have been fixed by past kernel commits (such as `cf3c4c03060b` [8] and `c9bfbb31af7c` [9]).

Buffer-destroy omission. The driver should destroy the created DMA buffer before removal; otherwise memory leaks will occur. Several existing approaches of resource-leak detection (such as Hector [48] and PR-Miner [32]) have found some such problems, and they have been fixed by past kernel commits (such as `37c85c3498c5` [10] and `7ca2cd291fd8` [11]).

Summary. Most of the above DMA problems are related to *DMA creation and destroy*, which are performed by calling specific DMA interfaces. Thus, most existing approaches check the rules of these DMA interfaces to detect DMA problems. In fact, besides calling these DMA interfaces, performing DMA accesses can also have security risks, which have not been fully realized by existing research. Thus, in this paper, we instead focus on detecting unsafe DMA accesses which are introduced in Section 2.3.

2.3 Security Risks of DMA Accesses

According to the type of DMA buffer introduced in Section 2.1, a DMA access can be a streaming DMA access or a coherent DMA access, which has different security risks:

Streaming DMA access. Once a streaming DMA buffer is mapped, it belongs to the hardware device instead of the CPU. Until the buffer has been unmapped, the driver should not access the content of the streaming DMA buffer; one exception is that the driver is allowed to access buffer content during synchronization with hardware registers and CPU cache [17]. Otherwise, accessing the content of the streaming DMA buffer can introduce data inconsistency between hardware registers and CPU cache, causing unexpected hardware behaviors. For short, we call such a problem as *inconsistent DMA access*.

Figure 2 presents a real inconsistent DMA access in the *rtl8192ce* wireless device driver in Linux 5.6. In the function `rtl92ce_tx_fill_cmddesc`, `pci_map_single` is called to map `skb->data` to a streaming DMA buffer on line 531. Then, the local variable `hdr` points to `skb->data` on line 535. After that, on line 536, `hdr->frame_control` is read and assigned to `fc`, namely a streaming DMA access is performed

```
FILE: linux-5.6/drivers/net/wireless/realtek/rtlwifi/rtl8192ce/trx.c
522. void rtl92ce_tx_fill_cmddesc(...) {
        ......
        // Streaming DMA mapping
531.    dma_addr_t mapping = pci_map_single(..., skb->data, ...);
        ......
535.    struct ieee80211_hdr *hdr = (struct ieee80211_hdr *)(skb->data);
536     fc = hdr->frame_control;  // Inconsistent DMA access!
        ......
584. }
```

Figure 2: Example inconsistent DMA access.

without synchronization, causing an inconsistent DMA access. The driver developers admit that this problem can cause unexpected hardware behaviors, which can make the driver crash. This problem was introduced in Linux 4.4 (released in Jan. 2016) and was fixed 4.5 years later (Oct. 2020) by us, based on a bug report of our approach SADA. We fixed this problem by accessing `hdr->frame_control` before calling `pci_map_single`.

Coherent DMA access. Different from streaming DMA buffers, coherent DMA buffers do not require explicit synchronization with hardware registers and CPU cache. But on one hand, because a hardware device can be malfunctioning or untrusted, it can write bad data into coherent DMA buffers; on the other hand, as the hardware device and driver can both modify the data in coherent DMA buffers, the driver may get different data when reading the same coherent DMA buffer, causing double-fetch cases. For the two reasons, the driver should perform correct validation of the data from DMA buffers before using it. Otherwise, security bugs (such as buffer overflow and invalid-pointer access) can be triggered. For short, we call such a problem as *unchecked DMA access*.

Figure 3 presents a confirmed unchecked DMA access in the *vmxnet3* network device driver in Linux 5.6. In the function `vmxnet3_probe_device`, `dma_alloc_coherent` is called to allocate a coherent DMA buffer assigned to `adapter->rss_conf`. In the function `vmxnet3_get_rss`, `adapter->rss_conf` is assigned to a local variable `rssConf`, and then `rssConf->indTableSize` is assigned to a local variable `n`. Thus, `n` stores the data in the coherent DMA buffer of `adapter->rss_conf`, and it can be modified to a bad value by the malfunctioning or untrusted device. In this case, `n` can be larger than the bound of `rssConf->indTable`, causing a buffer-overflow bug when `rssConf->indTable[n]` is read. This problem was introduced in Linux 3.16 (released in Aug. 2014) and was fixed nearly 6 years later (Jun. 2020) by us, based on a bug report of our approach SADA. We fixed it by adding a check of `n` with the bound of `rssConf->indTable` before `rssConf->indTable[n]` is read.

Rules of DMA accesses. For better understanding, we illustrate the rules of streaming and coherent DMA accesses with real DMA interfaces of the Linux kernel in Figure 4. The code segments shown in Figure 2 and Figure 3 obviously violate the rules, and thus they have *unsafe DMA accesses*.

```
FILE: linux-5.6/drivers/net/vmxnet3/vmxnet3_drv.c
3240. static int vmxnet3_probe_device(...) {
        ......
        // Coherent DMA allocation
3373.   adapter->rss_conf = dma_alloc_coherent(...);
        ......
3531. }
FILE: linux-5.6/drivers/net/vmxnet3/vmxnet3_ethtool.c
693. static int vmxnet3_get_rss(...) {
        ......
696.    struct UPT1_RSSConf *rssConf = adapter->rss_conf;
697.    unsigned int n = rssConf->indTableSize;
        ......
704.    while (n--)
705.       p[n] = rssConf->indTable[n];  // Possible buffer overflow
706.    return 0;
707. }
FILE: linux-5.6/drivers/net/vmxnet3/upt1_defs.h
80. struct UPT1_RSSConf {
81.     u16 hashType;
        ......
86.     u8 indTable[UPT1_RSS_MAX_IND_TABLE_SIZE]; // Bound is 128
87. }
```

Figure 3: Example unchecked DMA access.

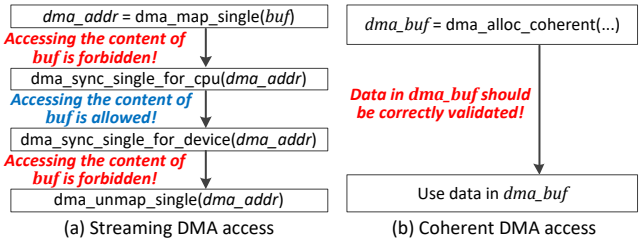

Figure 4: Rules of DMA accesses with typical interfaces.

2.4 Threat Model

Our threat model consists of an adversary that attacks the OS through DMA accesses by leveraging software defects in benign drivers to achieve malicious goals, such as denial of service and privilege escalation. In practice, a device driver is often provided by the OS kernel, and it is used to manage multiple third-party hardware devices and support specific workloads at the user level. For example, the USB core driver provided by the Linux kernel is used to control different kinds of USB devices and support user-level USB services. Thus, drivers are always considered to be benign, but hardware devices and user-level workloads can be untrusted. As a result, attacks can be launched in two ways. First, attackers can execute specific workloads at the user level to trigger inconsistent DMA accesses, which can cause unexpected hardware behaviors or even crash the OS. Second, attackers can use an untrusted hardware device to provide malicious data to the driver via DMA buffers, which can cause buffer overflow, invalid-pointer access, and other serious security issues.

2.5 State of the Art for DMA-Access Checking

Recently, several driver fuzzing approaches [45, 51, 52] have found some unchecked DMA accesses, by generating and mutating hardware inputs from simulated devices. But they may miss many real unchecked DMA accesses due to lim-

ited code coverage of runtime testing. Moreover, they do not consider the synchronization of streaming DMA buffers, and thus cannot detect inconsistent DMA accesses. Besides, they require associated simulated devices to run the tested drivers, but implementing such simulated devices often requires much manual work. To solve these limitations, we aim to design an effective static-analysis approach to automatically and accurately detect unsafe DMA accesses as many as possible.

2.6 Study of DMA in Linux Device Drivers

To understand the importance of detecting unsafe DMA accesses, we need to know how many existing device drivers have DMA operations. To find the answer, we manually study the driver source code in the Linux kernel, to calculate the proportion of device drivers that have DMA operations. Due to the large number of Linux device drivers and time constraints, we select all the drivers of 8 common classes in Linux 5.6 and manually read their source code, to identify the drivers that call DMA interfaces defined in the Linux kernel. Considering that a driver may be generated from multiple source files, we identify the number of drivers by manually checking their Makefiles in the Linux kernel.

Table 2 shows the results of our study. About 46% of studied drivers explicitly call DMA interfaces, indicating that DMA operations are common in existing device drivers. For this reason, it is important to check the safety of DMA accesses in device drivers.

Driver class	Source file (.c)	Driver number	Call DMA interface
Ethernet	1102	319	168 (53%)
Wireless	827	143	46 (32%)
Crypto	209	75	51 (68%)
GPU	2459	180	57 (32%)
MMC	119	95	43 (45%)
SCSI	414	153	87 (57%)
Infiniband	308	26	24 (92%)
USB	501	297	114 (38%)
Total	5939	1288	590 (46%)

Table 2: Study results of DMA in Linux device drivers.

From Figure 2 and Figure 3, we find an interesting characteristic of DMA-buffer creation, namely *when DMA buffers are created, they are often stored in data structure fields in the driver code*. The data structure fields `skb->data` in Figure 2 and `adapter->rss_conf` in Figure 3 are both such examples. This characteristic is understandable, because to pass key information (including DMA buffers) between different functions, device drivers often wrap such information in specific data structures and share them via function arguments. To clearly know whether this characteristic is common in existing drivers, we manually study the source code of 1288 drivers in Table 2 again. Specifically, we first identify the function calls to streaming DMA mapping and coherent DMA allocation, and then check whether the created DMA buffers are stored in data structure fields. Considering that a DMA buffer may be first stored in a local variable after creation (such as `p = dma_alloc_coherent(...)`) and then this local variable is assigned to a data structure field (such as `dev->dma_buf = p`), we also manually check the alias relationship between variables in the study.

Driver class	Struct / Streaming	Struct / Coherent	Struct / Both
Ethernet	490 / 563 (87%)	443 / 493 (90%)	933 / 1056 (88%)
Wireless	101 / 119 (85%)	90 / 103 (87%)	191 / 222 (86%)
Crypto	264 / 280 (94%)	90 / 93 (97%)	354 / 373 (95%)
GPU	41 / 49 (84%)	48 / 52 (92%)	89 / 101 (88%)
MMC	44 / 44 (100%)	13 / 13 (100%)	57 / 57 (100%)
SCSI	86 / 105 (82%)	382 / 487 (78%)	468 / 592 (79%)
Infiniband	43 / 60 (72%)	91 / 95 (96%)	134 / 155 (86%)
USB	23 / 27 (85%)	74 / 85 (87%)	97 / 112 (87%)
Total	1092 / 1247 (88%)	1231 / 1421 (87%)	2323 / 2668 (87%)

Table 3: Study results of data structure fields for DMA.

Table 3 shows the results of our study. About 87% of created DMA buffers are stored in data structure fields, which indicates that this characteristic is common in exsiting drivers. Inspired by this, we can first select data structure fields of DMA buffers and then use them to identify DMA accesses.

3 Challenges

To effectively detect unsafe DMA accesses via static analysis, we need to solve three main challenges:

C1: Identifying DMA accesses. In the driver code, each DMA access is implemented as a regular variable access, as shown in Figure 2 and Figure 3. Thus, compared to non-DMA hardware accesses calling specific kernel interfaces (such as `ioread8`), identifying DMA accesses is more difficult.

Figure 2 shows that the DMA-buffer creation and access are in the same function, namely in an *explicit control flow*. Thus, an intuitive solution is to first identify each DMA-buffer creation and then perform a flow-sensitive analysis starting from this creation to identify DMA accesses. But this solution is limited, as in many cases, the DMA-buffer creation and access are in two different functions without explicit execution order from static code observation, namely in a *broken control flow*. The code in Figure 3 is such an example. The coherent DMA buffer is allocated in the function `vmxnet3_probe_device`, and the buffer is accessed in `vmxnet3_get_rss`, but the two functions do not have explicit execution order from static code observation. Thus, identifying DMA accesses in the case of broken control flow is challenging.

C2: Checking the safety of DMA accesses. Given a DMA access, we need to check whether it is safe, according to the rules shown in Figure 4. Specifically, for a streaming DMA access, we need to check whether it occurs during the synchronization with hardware registers and CPU cache; for a coherent DMA access, we need to check whether it can cause possible insecure influence in driver code. To accurately and completely check the safety of these DMA accesses, we need

perform a flow-sensitive and inter-procedural analysis of the Linux kernel code. However, as the Linux kernel code base is very large (amounting to over 18M lines of code counted by CLOC [16] in Linux 5.6) and complex (involving lots of function calls), performing such analysis can be difficult.

C3: Dropping false positives. Without validating code-path feasibility, static analysis can report many false positives. Thus, many previous approaches [13, 14, 39, 47, 50] use SMT solvers to validate all encountered code paths during analysis. However, an SMT solver is often expensive and the Linux kernel code base is very large and complex, so scaling the validation of code-path feasibility in this way is challenging.

4 Key Techniques

To solve the above challenges, we propose three key techniques. For *C1*, we propose a field-based alias analysis to effectively identify DMA accesses, according to the information of DMA mapping creation. For *C2*, we propose a flow-sensitive and pattern-based analysis to accurately and efficiently check the safety of DMA accesses. For *C3*, we propose an efficient code-path validation method to drop false positives and reduce the overhead of using SMT solvers. We will introduce them as follows:

4.1 DMA-Access Identification

As we studied in Section 2.6, about 87% of created DMA buffers are stored in data structure fields. Inspired by this observation, we propose a field-based alias analysis to identify DMA accesses in the cases of broken control flow, by matching the data structure type and field of the created DMA buffers and DMA accesses. This analysis has two steps:

S1: Handling DMA-buffer creation. In this step, we identify each DMA-buffer creation and collect its data structure field. Figure 5 shows the procedure of this step. The set *dma_var_set* stores the variables of created DMA buffers, and it is used to identify DMA accesses in the cases of explicit control flow; the set *dma_struct_set* stores the information about data structure fields of created DMA buffers (including data structure type and field), and it is used to identify DMA accesses in the cases of broken control flow.

This step first initializes *dma_var_set* and *dma_struct_set* to empty, and then analyzes each function call in the driver code. This step checks whether the function call is used to create a DMA buffer, according to the name of its called function. If not, this call is neglected. Then, this step gets the variable *ret_var* to which the function call's return value is stored. Because *ret_var* may be aliased with other variables, this step performs an intra-procedural, flow-insensitive and Andersen-style alias analysis [1] to identify all variables aliased with *ret_var* (including *ret_var*). This step stores *ret_var* and its aliased variables in a set *var_set*. For each variable *var* in

```
GetDmaInfo: Get data structure fields of DMA buffers created in the driver
 1: dma_var_set := ø; dma_struct_set := ø;
 2: foreach call in driver do
 3:     if call is not used to create DMA buffers then
 4:         continue;
 5:     end if
 6:     ret_var := GetStoredReturnVal(call);
 7:     var_set := GetAliasVarSet (ret_var); // Including ret_var
 8:     foreach var in var_set do
 9:         AddSet(var, dma_var_set);
10:         struct_info := GetStructInfo(var); // Get structure type and field
11:         if struct_info != null then
12:             AddSet(struct_info, dma_struct_set);
13:         end if
14:     end foreach
15: end foreach
```

Figure 5: Procedure of handling DMA-buffer creation.

var_set, this step adds it into *dma_var_set*, and gets its data structure information <*struct_type, field*> (including the data structure type and field) to store in *struct_info*. If *struct_info* is non-null, namely *var* is a data structure field, this step adds *struct_info* into *dma_struct_set*.

S2: Identifying DMA accesses. Because we have already collected the variables and data structure information of DMA buffers, the idea of this step is to identify which variable accesses involve these variables or match the data structure information, and such variable accesses are identified as DMA accesses. Specifically, according to the two sets *dma_var_set* and *dma_struct_set* collected in *S1*, our field-based analysis identifies DMA accesses in the driver. Figure 6 shows the procedure of this step. The set *dma_access_set* stores all identified DMA accesses.

```
GetDmaAccess: Identify DMA accesses in the driver
 1: dma_access_set := ø;
 2: foreach inst in driver do
 3:     if inst is not a load or store instruction then
 4:         continue;
 5:     end if
 6:     var := GetAccessedVar(inst);
 7:     struct_info := GetStructInfo(var); // Get structure type and field
 8:     if CheckItemInSet(var, dma_var_set) == true or
 9:        CheckItemInSet(struct_info, dma_struct_set) == true then
10:         var_set := GetAliasVarSet(var); // Including var
11:         inst_set := GetInstSetFromVarSet(var_set); // Including inst
12:         AddSetInSet(inst_set, dma_access_set);
13:     end if
14: end foreach
```

Figure 6: Procedure of identifying DMA access.

This step first initializes *dma_access_set* to empty, and then analyzes each instruction in the driver code. This step checks whether the instruction is a load (read) or store (write) instruction. If not, this instruction is neglected. Then, this step gets the variable *var* accessed by the instruction, and gets its data structure information <*struct_type, field*> (including data structure type and field) to store in *struct_info*. After that, this step checks whether *var* is in *dma_var_set* (for the case of

explicit control flow) or *struct_info* is in *dma_struct_set* (for the case of broken control flow). If so, this step again uses the alias analysis mentioned in *S1* to identify all variables aliased with *var* (including *var*), then gets the instructions accessing the identified variables, and finally adds these instructions into *dma_access_set*.

Alias analysis. Note that performing alias analysis in *S1* and *S2* is necessary, because the variables of DMA buffers may be aliased with other variables. For this reason, identifying these aliased variables is helpful to reducing false negatives of DMA-access identification.

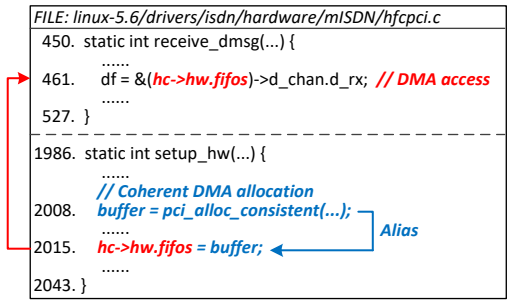

Figure 7: Example of identifying DMA access.

Example. We use the Linux *hfcpci* driver code to illustrate our field-based alias analysis. First, in the function `setup_hw`, *S1* finds that the function call to `pci_alloc_consistent` creates a DMA buffer on line 2008, and thus *S1* records the variable `buffer` that stores the return value of this function call. Then, *S1* looks for the variables aliased with `buffer`, and finds that `hc->hw.fifo` is such a variable, from the assignment on line 2015. Thus, *S1* records the data structure type and field of `hc->hw.fifo`. After that, *S2* identifies the DMA accesses by matching the pairs of data structure type and field collected in *S1*. In the function `receive_dmsg`, *S2* finds that the accessed variable on line 461 matches the pair of data structure type and field that *S1* collects on line 2015. Thus, *S2* identifies this variable access is a DMA access.

4.2 DMA-Access Safety Checking

Based on the DMA accesses identified in Section 4.1, we use a flow-sensitive and pattern-based analysis to check the safety of DMA accesses and detect possible unsafe DMA accesses. We perform the analysis using different patterns for streaming DMA accesses and coherent DMA accesses:

Checking streaming DMA access. We check whether each streaming DMA access is performed: 1) between DMA mapping and unmapping and 2) during DMA-buffer synchronization with hardware and CPU (e.g. whether it is performed between the function calls to `dma_sync_single_for_cpu` and `dma_sync_single_for_device` shown in Figure 4(a)). If not, we report a possible inconsistent DMA access. However, because DMA mapping, DMA-buffer synchronization and DMA unmapping may not have explicit execution order

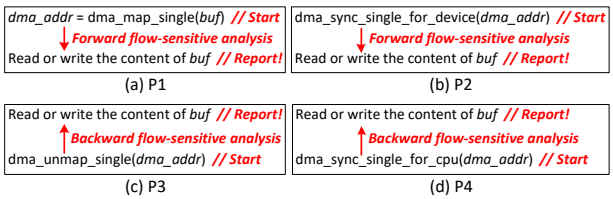

Figure 8: Patterns for checking streaming DMA access.

by statically observing the driver code (namely in broken control flow), checking streaming DMA access can be difficult. To solve this problem, we perform checking according to four patterns, which are illustrated in Figure 8 using the DMA interfaces in Figure 4(a):

P1 and P2: We perform a forward flow-sensitive analysis starting from a function call to DMA mapping or DMA synchronization for hardware device, and report an inconsistent DMA access when a DMA access is performed and no other DMA operations occur in the analyzed code path.

P3 and P4: We perform a backward flow-sensitive analysis starting from a function call to DMA unmapping or DMA synchronization for CPU, and report an inconsistent DMA access when a DMA access is performed and no other DMA operations occur in the analyzed code path.

Note that the forward and backward flow-sensitive analyses are both inter-procedural, to detect deep inconsistent DMA accesses crossing function calls. Besides, to improve efficiency, the two analyses never validate path conditions, and just record the basic blocks in each code path for validating code-path feasibility in Section 4.3.

Checking coherent DMA access. We check whether the data read from a coherent DMA buffer can cause possible insecure influence in driver code. Specifically, considering that infinite looping, buffer overflow and invalid-pointer access are three typical kinds of security problems that can be triggered by problematic hardware accesses [27], we focus on the related patterns in our safety checking:

P1) Infinite looping: affecting loop condition in loop iteration. For a given loop, a variable checked in the loop condition can be affected by the data from a coherent DMA access that is performed in the loop iteration. In this case, the malfunctioning or untrusted hardware device can modify the corresponding DMA buffer in each iteration, to change the variable in the loop condition and cause infinite loop polling. Figure 9(a) shows such an example in the *iwlwifi* driver in Linux 5.6. In the function `iwl_pcie_alloc_ict`, a coherent DMA buffer is allocated and it is stored in the variable `trans_pcie->ict_tbl`. In the function `iwl_pcie_int_cause_ict`, an element of `trans_pcie->ict_tbl` is accessed and assigned to a variable `read` in the loop iteration. In the loop condition, `read` is compared with zero. In this example, the malfunctioning or untrusted hardware device can always set the accessed element of `trans_pcie->ict_tbl` to be non-zero, to make the loop infinitely run.

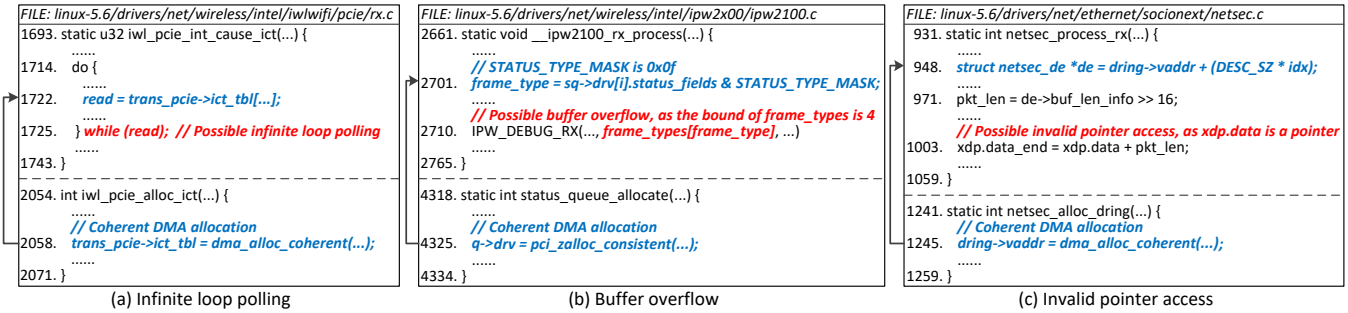

Figure 9: Example of patterns for checking coherent DMA access.

P2) Buffer overflow: affecting an array index. To access an element of a given array, the driver needs to use a variable as the array index, and this variable can be affected by the data from a coherent DMA access. In this case, the malfunctioning or untrusted hardware device can modify the DMA buffer, to make the variable of array index larger than the array bound, causing a buffer-overflow bug. Figure 9(b) shows such an example in the *ipw2x00* driver in Linux 5.6. In the function `status_queue_allocate`, a coherent DMA buffer is allocated and it is stored in the variable `q->drv`. In the function `__ipw2100_rx_process`, the data stored in the DMA buffer `sq->drv[i].status_field` is anded with `STATUS_TYPE_MASK` (0x0f), and then the result is assigned to a variable `frame_type`. After that, `frame_type` is used as the index to access the array `frame_types` whose bound is 4. In this example, the malfunctioning or untrusted hardware device can modify `sq->drv[i].status_field` into a bad value (such as 8), to make `frame_type` larger than the bound of `frame_types` even though being anded with `STATUS_TYPE_MASK`. Thus, a buffer-overflow bug can be triggered when `frame_types[frame_type]` is accessed.

Note that the buffer-overflow bug shown in Figure 3 is also such an example, but the two bugs in these two figures are a little different. The bug in Figure 3 lacks a mask or check operation on the array index, while the bug in Figure 9(b) has a mask operation but this operation is incorrect. Inspired by Figure 9(b), we also need to check whether the validation operation (mask or check) on the array index in the driver code is correct, if this operation exists.

P3) Invalid-pointer access: affecting the offset of an accessed pointer. The driver often uses a variable as the offset to access a pointer, and this variable can be affected by the data from a coherent DMA access. In this case, the malfunctioning or untrusted hardware device can modify the DMA buffer, to change the offset variable into a bad value and make the accessed pointer invalid. Figure 9(c) shows such an example in the *socionext* driver in Linux 5.6. In the function `netsec_alloc_dring`, a coherent DMA buffer is allocated and it is stored in the variable `dring->vaddr`. In the function `netsec_process_rx`, the data stored in the DMA buffer `de->buf_len_info` is right shifted with 16, and then the result is assigned to a variable `pkt_len`. After that, `pkt_len` is used as the offset to access the pointer based on `xdp.data`. In this example, the malfunctioning or untrusted hardware device can modify `de->buf_len_info` into a bad value (such as 0xffff), to make `pkt_len` very large. Thus, the pointer `xdp.data + pkt_len` can be invalid for the driver to access.

TaintAnalysis: Identifying the variables affected by a cohenret DMA access *"dma_access"* in the code path *"code_path"*

```
1: var_set := ø; inst_set := ø;
2: taint_var := GetAccessedVar(dma_access);
3: AddSet(taint_var, var_set);
4: AddSet(dma_access, inst_set);
5: foreach inst in GetInstSetInPath(code_path) do
6:     res_var := GetResultVal(inst);
7:     op_var_set := GetOperandVal(inst);
8:     if GetIntersect(op_var_set, var_set) != ø then
9:         AddSet(inst, inst_set);
10:        if res_var != null then
11:            AddSet(res_var, var_set);
12:        end if
13:    end if
14: end foreach
```

Figure 10: Procedure of taint analysis.

Taint analysis. In the above patterns, locating the variables affected by the data from a given coherent DMA access is an important task. To finish this task, we use a flow-sensitive and inter-procedural taint analysis to identify such variables. Figure 10 shows the procedure of this taint analysis. It starts from each coherent DMA access and forwardly analyzes each instruction in the code path. The taint analysis maintains two sets, namely *var_set* to store variables affected by the DMA access and *inst_set* to store instructions containing these affected variables. The analysis first initializes *var_set* and *inst_set* to empty. Then, the analysis gets the accessed variable of the DMA access and adds it into *var_set*, and the analysis also adds this DMA access into *inst_set*. After that, the analysis handles each instruction *inst* in the code path. For *inst*, it gets the result variable *res_var* and the set of operand variables *op_var_set*. The analysis checks whether *op_var_set* and *var_set* have intersection, namely whether *inst* has an operand affected by the DMA access. If so, the analysis adds *inst* into *inst_set*, and adds *res_var* into *var_set* if *res_var* is non-null (an instruction may not have a result variable, and thus *res_var* of this instruction is null).

Note that to improve efficiency, this taint analysis never validates path conditions and just records the basic blocks in each code path for validating code-path feasibility in Section 4.3. Besides, to avoid infinite looping on recursive calls in the code path, the analysis records the analyzed basic blocks and ends when encountering a basic block within a loop that has been handled.

4.3 Code-Path Validation

Given the possible unsafe DMA accesses (their code-path feasibility is not validated) found in Section 4.2, we validate their code paths using an SMT solver Z3 [66] to drop false positives. Compared to the traditional strategy of validating all code paths during static analysis, our strategy is more efficient, as we believe that *the code paths containing possible unsafe DMA accesses often occupy a very small proportion of all code paths*. Thus, our strategy can reduce much unnecessary validation of code paths that unlikely to contain unsafe DMA accesses. Our code-path validation has three steps:

S1: Getting path constraints. We translate each instruction into a constraint using the Z3 grammar. Specifically, for each assignment instruction (such as $a = b + c$), we translate it into an equation constraint (such as $a == b + c$); for each branch condition (such as *if* $(a > b)$), we translate it into a constraint according to the successor basic block of this condition from the given code path (such as $a > b$ if the successor basic block is in the true branch or $a \leq b$ if the the successor basic block is in the false branch).

S2: Adding constraints for triggering security bugs. As shown in Figure 9(b), a driver may have validation (mask or check) operations on the variable affected by the DMA access, to prevent the related security bug being triggered. If such validation operations are ignored, many false unsafe DMA accesses may be reported. To reduce such positives, for each possible unsafe DMA access, we add proper constraints for triggering the related security bug. For example in Figure 9(b), we add a constraint *"frame_type > 4"* to indicate that the buffer-overflow bug can occur when `frame_type` is larger than the bound of the array `frame_types`. Note that not all unsafe DMA accesses require such extra constraints, such as inconsistent DMA accesses. Thus, for such an unsafe DMA access, we just add an empty constraint.

S3: Solving all constraints. We put the path constraints produced in *S1* and the additional constraints for triggering the related security bug produced in *S2* into the SMT solver Z3, to check whether each possible unsafe DMA access can occur. If these constraints cannot be satisfied, we consider this possible unsafe DMA access as a false positive and drop it.

5 SADA Approach

Based on the three key techniques in Section 4, we propose a static-analysis approach named SADA, to automatically and accurately detect unsafe DMA accesses in device drivers. We have implemented SADA using Clang-9.0 [15] and performed static analysis on the driver LLVM bytecode files. SADA works automatically without manual effort, given the source files of device drivers. Figure 11 shows the overall architecture of SADA.

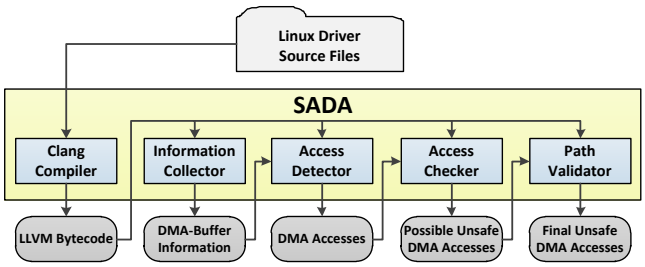

Figure 11: SADA architecture.

Based on this architecture, SADA consists of four phases:

P1: Code compilation. The *Clang compiler* compiles each driver source file into an LLVM bytecode file. Because multiple functions defined in different device drivers may share the same function name, inter-procedural analysis may identifies incorrect functions to analyze. To solve this problem, during linking, SADA records the set of source files generating the same driver. According to this information, SADA can accurately select correct functions when performing inter-procedural analysis.

P2: DMA-access identifying. The *information collector* collects the information about each DMA-buffer creation from each LLVM bytecode file, and then the *access detector* uses this information and performs our field-based alias analysis to identify DMA accesses.

P3: DMA-access checking. The *access checker* uses our flow-sensitive and pattern-based analysis to check the safety of each identified DMA access and detect possible unsafe DMA accesses.

P4: Unsafe-DMA-access validating. The *path validator* uses our code-path validation method to drop false unsafe DMA accesses. Besides, our flow-sensitive analysis may find many repeated unsafe DMA accesses when their DMA-buffer creation and DMA accesses are identical but only differ in their code paths. To solve this problem, the path validator also drops repeated results according to the positions of DMA-buffer creation and DMA access. Finally, SADA generates readable reports of final unsafe DMA accesses.

6 Evaluation

To validate the effectiveness of SADA, we evaluate it on the driver code of Linux 5.6. We run the evaluation on a regular x86-64 desktop with eight Intel i7-3770 CPU@3.40GHz processors and 16GB physical memory. We use the kernel configuration *allyesconfig* to enable all kernel code for the x86-64 architecture.

Description		SADA
Code analysis	Source files (analyzed / all)	14.6K / 17.9K
	Source code lines (analyzed / all)	8.8M / 10.3M
	Handled unique functions	334.7K
	Handled unique LLVM instructions	33.0M
DMA-access identifying	Encountered DMA-buffer creation	2,781
	DMA buffers in data structure fields	2,074
	Identified DMA accesses	28,732
DMA-access checking	Dropped DMA accesses (false + repeated)	736 (251 + 485)
	Unsafe DMA accesses (real / all)	284 / 321
	Inconsistent DMA accesses (real / all)	123 / 131
	Unchecked DMA accesses (real / all)	161 / 190
Time usage	DMA-access identification	62m
	DMA-access checking	208m
	Total time	270m

Table 4: Detection results of the Linux 5.6 driver code.

6.1 Detection of Unsafe DMA Accesses

We run SADA to automatically check the Linux driver source code (in the *drivers* and *sound* directories), and then manually check all the unsafe DMA accesses found by SADA. Table 4 summarizes the results, and source code lines are counted by CLOC [16]. From Table 4, we have the following findings:

Analyzing driver code. SADA can scale to a large code base. It spends 270 minutes on analyzing 8.8M lines of source code in 14.6K source files. The remaining 1.5M lines of source code in 3.3K source files are not analyzed, because they are not enabled by *allyesconfig* for the x86-64 architecture.

Identifying DMA accesses. SADA is effective in identifying DMA accesses in device drivers. It identifies over 28K DMA accesses according to 2,781 created DMA buffers. Among these DMA buffers, SADA identifies that 75% of them are stored in data structure fields. This percentage is a little lower than the manual study result (87%) in Section 2.6, as SADA still fails to identify some data structure fields of created DMA buffers in complex cases. For example, some drivers use their own wrapper functions that call primitive DMA kernel interfaces to create DMA buffers, but SADA only identifies DMA-buffer creation by looking for primitive DMA kernel interfaces, without analyzing these wrapper functions. Even so, the result (75%) here again proves that most of created DMA buffers are stored in data structure fields.

Detecting unsafe DMA accesses. SADA finds 321 unsafe DMA accesses, including 131 inconsistent DMA accesses and 190 unchecked DMA accesses. Among them, we identify that 284 are real, including 123 inconsistent DMA accesses and 161 unchecked DMA accesses. The false positive rate is only 11.5%, which benefits from our key techniques, such as field-based analysis for DMA-access identification and code-path validation. Specifically, SADA drops 251 false unsafe DMA accesses and 485 repeated unsafe DMA accesses. We have reported the 284 real unsafe DMA accesses to Linux kernel developers, and 105 of them have been confirmed. We are still waiting for the response of remaining ones.

Result distribution. We classify the 284 real unsafe DMA accesses and 105 confirmed ones according to driver class. Table 5 shows the distribution results. We find that network and SCSI drivers share 71% of the real unsafe DMA accesses and 66% of the confirmed ones.

Driver class	Network	SCSI	Crypto	USB	Others
Defects	113 (40%)	89 (31%)	41 (14%)	10 (4%)	31 (11%)
Confirmed	50 (48%)	19 (18%)	17 (16%)	8 (8%)	11 (10%)

Table 5: Distribution of found unsafe DMA accesses.

```
FILE: linux-5.6/drivers/thunderbolt/ctl.c
308. static struct ctl_pkg *tb_ctl_pkg_alloc(...) {
       ......
       // Coherent DMA allocation
314.   pkg->buffer = dma_pool_alloc(...);
       ......
321. }
FILE: linux-5.6/drivers/thunderbolt/icm.c
269. static bool icm_copy(...) {
270.   const struct icm_pkg_header *hdr = pkg->buffer;
271.
272.   if (hdr->packet_id < req->npackets) {
273.     size_t offset = hdr->packet_id * req->response_size;
274.
275.     memcpy(req->response + offset, pkg->buffer,
276.                                    req->response_size);
277.   }
278.   return hdr->packet_id == hdr->total_packets - 1;;
279. }
```

Figure 12: An unsafe DMA access in the *thunderbolt* driver.

Security impact of unsafe DMA accesses. The 123 inconsistent DMA accesses can cause unexpected hardware behaviors, making the driver crash and leading to DoS attacks. Among the 161 unchecked DMA accesses, 121 of them can trigger buffer-overflow bugs, causing memory overwrite or overread; 36 of them can trigger invalid-pointer accesses, causing arbitrary read or write, which can be exploited for privilege escalation; 4 of them can trigger infinite looping, causing DoS attacks.

Case study. Figure 12 presents an unsafe DMA access found by SADA, and it has high security impact. In the function `tb_ctl_pkg_alloc`, `dma_pool_alloc` is called to allocate a coherent DMA buffer that is stored in the variable `pkg->buffer`. In the function `icm_copy`, `pkg->buffer` is assigned to a variable `hdr`, and then the multiplication result of `hdr->packet_id` and `req->response_size` is stored in `offset`. After that, the function `memcpy` is called to copy the data stored in the DMA buffer `pkg->buffer` into the kernel memory buffer `req->response+offset`. In this example, if the hardware device is untrusted, the content of `pkg->buffer` and `hdr` can be modified by an attacker. By modifying the data of `hdr->packet_id`, the attacker can make the *if* statement on line 272 enter the true branch. Then, the attacker can again modify `hdr->packet_id` to a bad value larger than `req->npackets`, to access the kernel memory of an invalid pointer when `memcpy` is called on line 275. At this time, the attacker can inject malicious data into `pkg->buffer`, to make `memcpy` copy the malicious data into a confidential kernel memory area via this invalid pointer. In this example, `hdr->packet_id` is read twice, but its data can be modified by the untrusted hardware device before being read at the second time, causing a double-fetch situation.

6.2 False Positives and Negatives

False positives. SADA still reports 37 false unsafe DMA accesses, which are introduced for two main reasons:

First, the alias analyses used in SADA can make mistakes. On one hand, to identify DMA accesses in the cases of broken control flow, SADA uses a field-based analysis by only considering data field structures and fields, but it neglects alias relationship in the code path. On the other hand, to identify the variables aliased with a variable of the DMA buffer, SADA uses an intra-procedural and flow-insensitive alias analysis, which can be inaccurate due to ignoring flow sensitivity and missing inter-procedural analysis. For these reasons, SADA can identify false DMA accesses.

Second, although SADA uses Z3 to validate the path feasibility of unsafe DMA accesses, it can still make mistakes in some complex cases, such as complicated arithmetic in branch conditions and data dependence across function calls.

False negatives. SADA may still miss some real unsafe DMA accesses for four main reasons:

First, as describe in Section 6.1, SADA still fails to identify some data structure fields storing created DMA buffers, when the driver uses its own wrapper functions that call primitive DMA kernel interfaces to create DMA buffers. Thus, SADA can fail to identify and check the DMA accesses related to such DMA buffers.

Second, SADA does not analyze function-pointer calls when checking the safety of DMA accesses, and thus it cannot build complete call graphs for inter-procedural analysis. As a result, it can miss real unsafe DMA accesses involving the code reached through function-pointer calls.

Third, SADA neglect driver concurrency in DMA-access checking, and thus it can miss real unsafe DMA accesses, when DMA-buffer creation/synchronization and DMA accesses are performed in two concurrently-executed functions.

FILE: linux-5.6/sound/soc/fsl/fsl_dma.c

```
383. static int fsl_dma_open(...) {
        ......
        // Coherent DMA allocation
418.    dma_private = dma_alloc_coherent(...);
        ......
431.    dma_private->irq = dma->irq;
        ......
        // May affect kernel functionality
436.    ret = request_irq(dma_private->irq, ...);
        ......
509. }
```

Figure 13: A real unchecked DMA access of other patterns.

Finally, SADA only checks the safety of coherent DMA accesses according to three typical patterns that can trigger security bugs, as described in Section 4.2. In fact, coherent DMA accesses can be vulnerable in other patterns. Figure 13 shows such an example. In the function `fsl_dma_open`, `dma_alloc_coherent` is called to allocate a coherent DMA buffer and it is stored in the variable `dma_private`. Then, an interrupt line number `dma->irq` is stored in the DMA buffer `dma_private->irq`. After that, the kernel interface `request_irq` is called to register an interrupt handler with the interrupt line number stored in `dma_private->irq`. In this example, the malfunctioning or untrusted hardware device can modify `dma_private->irq` into a bad value that is unequal to `dma->irq`, which can affect the kernel functionality of registering interrupt handler. When implementing SADA, we tried to support this pattern in detecting unchecked DMA accesses, but selecting important kernel interfaces like `request_irq` requires much manual work and kernel experience, which damages the automation of SADA. Thus, at present, SADA does not use this pattern by default.

Type	Root cause	Number
Inconsistent DMA access	Access after mapping	108
	Incorrect synchronization	15
Unchecked DMA access	Missing safety check	134
	Wrong mask operation	5
	Bypassing check (double fetch)	22 (16)

Table 6: Root causes of unsafe DMA accesses.

6.3 Root Causes and Fixing Suggestions

We manually check the root causes of the 284 real unsafe DMA accesses by reviewing the driver code, and summarize the results in Table 6.

For the 123 inconsistent DMA accesses, we find two root causes of them:

(1) 113 are caused by accessing a DMA buffer after it is mapped without involving any synchronization (the inconsistent DMA access shown in Figure 2 is such an example). To fix them, we suggest performing DMA access before the DMA mapping. Interestingly, from the driver developers' replies to our bug reports, several of them even admit that they were unaware of the rules about DMA-buffer synchronization.

(2) 10 are caused by incorrect DMA-buffer synchronization, namely the DMA access occurs before the synchronization for hardware device or after the synchronization for CPU. To fix them, we suggest performing DMA access between the synchronization for hardware device and CPU.

For the 161 unchecked DMA accesses, we infer three root causes of them:

(1) 134 are caused by missing a safety check of the related array index or pointer offset affected by the DMA access (the unchecked DMA accesses shown in Figure 3 and Figure 9(c) are such examples). To fix them, we suggest adding a correct safety check of the related array index or pointer offset.

(2) 5 are caused by a wrong mask operations of the related array index affected by the DMA access (the unchecked DMA access shown in Figure 9(b) is such an example). To fix them, we suggest correcting the related mask operation.

(3) 22 are caused by the case that the safety check of the related array index affected by the DMA access can be bypassed. Specifically, this root cause can be further classified

into two cases. First, the safety check is not strong enough, and thus it can be bypassed by a bad value of corner cases to trigger security bugs. To fix such unchecked DMA accesses, we suggest enforcing the related safety check to avoid all possible corner cases. Second, for a given DMA buffer, its data is first validated in a safety check and then this DMA buffer is accessed again in the safe branch (the unchecked DMA access shown in Figure 12 is such an example). In this case, an attacker can use untrusted hardware device to modify the DMA buffer gain in the safe branch, causing double-fetch situation. To fix such unchecked DMA accesses, we suggest that the driver should first store the data of the DMA buffer into a local variable in kernel memory, and then check and access this local variable.

From the three above results, we find that most of the unchecked DMA accesses are caused by the first root cause. It indicates that many driver developers may be unaware that hardware devices can be untrusted and provide bad data. For the remaining two root causes, the related driver developer may be aware that hardware devices can provide bad data, but their implemented validation code of the data from DMA access is incorrect or weak. As a result, the attacker can still inject bad data into DMA buffers via untrusted hardware devices, to trigger serious security problems.

7 Discussion

7.1 Comparison to Existing Approaches

To our knowledge, SADA is the first approach that systematically detects unsafe DMA accesses, and thus we focus on comparing SADA to existing approaches that can check hardware accesses in device drivers.

Dynamic analysis. Several recent fuzzing approaches [45, 51, 52] have found some security bugs caused by the bad data from DMA buffers (unchecked DMA accesses). Different from these approaches, SADA can automatically cover much more driver code without executing test cases or preparing simulated devices. Thus SADA can find many real unchecked DMA accesses missed by them. Moreover, SADA can also find inconsistent DMA accesses that these approaches are unable to detect.

Static analysis. Generic static analysis frameworks (such as DR. CHECKER [38], Coccinelle [30] and Clang Static Analyzer [19]) can detect multiple types of OS bugs. Some of them (such as Coccinelle [30]) can check the calls to specific kernel interfaces about hardware inputs. However, each DMA access is implemented as a regular variable access, instead of calling specific kernel interfaces, and thus they fail to detect unsafe DMA accesses. By contrast, SADA uses a new field-based analysis to effectively identify DMA accesses and thus can accurately detect unsafe DMA accesses.

In addition, compared to DR. CHECKER [38] that also statically checks driver code, SADA has some differences in the implementation of driver code analysis. First, the taint analysis of DR. CHECKER uses the arguments of entry functions as taint sources, and it considers points-to relationships at each program point to support multiple bug checkers; while the taint analysis of SADA uses the variables of DMA accesses as taint sources, and it checks DMA-related operations in each code path without considering points-to relationships. Second, DR. CHECKER assumes that all kernel API functions are correctly used; while SADA does not have such assumption and it checks DMA-related API calls to detect misuses. Finally, DR. CHECKER does not check path feasibility during code analysis; while SADA uses an SMT solver to validate the code-path feasibility of each possible unsafe DMA access, to reduce false positives.

Carburizer [27] is a specific approach that detects and tolerates driver failures caused by malfunctioning hardware devices. It statically analyzes the driver code to check whether the data read from hardware registers is correctly validated before being used and can trigger reliability or security problems (such as infinite polling and buffer overflow), because hardware devices can fail and provide problematic inputs for drivers. To identify the variables affected by each hardware-register access, Carburizer uses a static taint analysis starting from each call to hardware-access kernel interfaces (such as `ioread8`), which is similar to the safety checking of DMA accesses in SADA. However, SADA has some important differences from Carburizer:

First, Carburizer cannot handle DMA accesses, because it relies on specific kernel interfaces to identify hardware-register accesses, but each DMA access is implemented as a regular variable access, instead of calling specific kernel interfaces; SADA uses a field-based alias analysis to effectively identify DMA accesses, according to the information of DMA-buffer creation.

Second, besides using static taint analysis to check the data read from hardware devices, SADA also uses a forward and a backward flow-sensitive analyses to check the context of streaming DMA accesses for detecting inconsistent DMA accesses, which Carburizer does not consider.

Finally, Carburizer uses a flow-sensitive analysis to check hardware-register accesses without validating code-path feasibility, which can introduce some false positives; SADA uses an SMT solver to accurately validate the code paths of unsafe DMA accesses after flow-sensitive analysis, in order to reduce false positives.

7.2 Limitations

SADA still has some limitations in detecting unsafe DMA accesses. First, SADA uses an intra-procedural and flow-insensitive alias analysis to identify the variables aliased with each variable of the DMA buffer. But due to ignoring flow sensitivity and missing inter-procedural analysis, this alias analysis can be inaccurate in complex cases, causing false pos-

itives in DMA-access identification and checking. To address this limitation, we can refer to existing approaches [25, 54] to perform inter-procedural and flow-sensitive alias analysis, in order to improve the accuracy of code analysis. Second, SADA does not analyze function-pointer calls in DMA-access checking at present, and thus it may miss real unsafe DMA accesses involving the code reached through function-pointer calls. To address this limitation, we can apply existing function-pointer analysis [3, 34] to enhance inter-procedural analysis in SADA. Finally, SADA does not consider driver concurrency in DMA-access checking at present, which can cause false negatives when DMA-buffer creation/synchronization and DMA accesses are performed in two concurrently-executed functions. To address this limitation, we can borrow existing concurrency analysis [2, 59] to check DMA accesses involving driver concurrency.

7.3 Exploitability of Unsafe DMA Accesses

Once knowing an unchecked DMA access, attackers can just inject malicious data to DMA buffers via untrusted hardware devices. When malicious data is used in critical control flow or data flow (such as an index into a buffer), serious security issues such as buffer overflow and invalid-pointer access can be triggered. Attackers just need to figure out how the data in a DMA buffer is used and what malicious data to inject in the DMA buffer. Thus, the exploitation is actually easier than that for traditional memory bugs caused by user-level inputs. Once knowing an inconsistent DMA access, attackers can execute specific workloads at the user level to trigger this defect, which can cause unexpected hardware behavior. In addition, if the data affected by this defect is used in critical control flow or data flow, it can also cause serious consequences like privilege escalation, which is analogous to race conditions.

7.4 Double Fetch Caused by DMA Access

Double fetch is a special situation that propably triggers security bugs. In this situation, the kernel reads the same variable twice and assumes its value should be unchanged. However, this assumption can be invalid when the value can be changed at runtime by some means. Existing approaches [49, 60–62, 64] focus on double-fetch situations caused by untrusted user-space memory.

However, as shown in Section 6.3, we find that DMA accesses to untrusted hardware devices can also cause double-fetch situations triggering security bugs (such as buffer overflow and invalid-pointer access). The unsafe DMA access shown in Figure 12 is such an example. For this reason, double-fetch situations caused by untrusted hardware devices should receive significant attention to avoid security bugs. We believe that SADA is applicable to detecting general double-fetch situations caused by DMA accesses, by adding more patterns in the safety checking of DMA accesses.

7.5 Avoiding Unsafe DMA Accesses

In this paper, we find that there are many unsafe DMA accesses in Linux driver code, and they can cause serious security problems. Thus, it is meaningful to discuss how to avoid unsafe DMA accesses when implementing a new device driver. We have three suggestions about it:

First, if the data stored in a DMA buffer needs to be accessed by the driver for multiple times, we suggest the driver to use a coherent DMA buffer here. In this way, there is no need to perform explicit synchronization operation for the DMA buffer with the CPU cache and hardware registers, which can avoid introducing inconsistent DMA accesses.

Second, we find that all unchecked DMA accesses are caused by the fact that the data from related DMA buffers affects the driver's data flow or control flow. Thus, if the untrusted hardware device injects bad data in these DMA buffers, the driver execution will be affected, increasing the possibility of triggering security problems. To avoid this case, we suggest the driver not to access the data stored in DMA buffers and just to transfer this data to user-space memory or hardware registers. In fact, many existing drivers use DMA only for data transfer. For example, many network device drivers use DMA buffers only to store data packets from/to network devices, and just transfer these data packets to/from user-space memory without accessing them. But some drivers have to access the data stored in DMA buffers, as they require hardware information (such as device descriptors) to control code execution. For these drivers, they should carefully validate the data from DMA buffers.

Finally, to avoid double fetch caused by DMA access, we suggest the driver to use a local variable to store the data from the DMA access, and access this variable instead of performing the same DMA access for multiple times.

8 Related Work

8.1 Static Analysis for Kernel Security

To enhance kernel security, many existing approaches use static analysis to check OS kernel code.

Analyzing security check. An OS kernel has many security checks to validate data correctness. If necessary security checks are missing or incorrect, serious security bugs (such as buffer-overflow bugs and null-pointer dereferences) can occur. To analyze security checks and detect related security bugs, some approaches [35, 36, 63, 67] have been proposed. CRIX [36] is a practical approach for detecting missing-check bugs in OS kernels with an inter-procedural, semantic- and context-aware analysis. These approaches focus on security checks about the return values or parameters of function calls, but cannot handle the data from DMA accesses, as each DMA access is implemented as a regular variable access, instead of calling specific functions.

Detecting API misuse. In an OS kernel, there are many API rules, and violating these rules can cause serious security bugs (such as resource leaks and double locks). To detect API misuses, some approaches [3, 4, 30, 39] use known API rules to check the kernel code. Besides, to learn implicit API rules, some approaches [5, 20, 31, 32, 48] perform specification mining by analyzing the kernel code. Different from API misuse, unsafe DMA access not only involves the call to DMA kernel interfaces but also involves the variable access to related DMA buffer.

Checking untrusted access. User-space memory is considered to be untrusted for OS kernels, and thus the kernel needs to carefully check the data from user-space memory; otherwise security problems (such as privilege escalation and information leakage) can occur. To detect these problems, some approaches [26, 49, 60–64] have been proposed. Besides, some researchers also realized that hardware devices can be malfunctioning or untrusted to affect kernel security, and thus they have proposed several approaches [27, 37] to detect unsafe hardware accesses. But they cannot check DMA accesses, as they rely on specific kernel interfaces to identify hardware accesses, but each DMA access is implemented as a regular variable access, instead of calling kernel interfaces.

8.2 Kernel Fuzzing

Fuzzing is a popular technique to improve code coverage in runtime testing. Many kernel fuzzing approaches have been proposed and shown promising results in detecting bugs.

Fuzzing system calls. Most kernel fuzzing approaches [18, 24, 43, 56, 57] focus on mutating and generating system calls, to test whether the kernel can correctly handle these system calls. MoonShine [43] analyzes the code-coverage contribution and dependencies of provided system calls from their traces, to select effective seeds for subsequent mutation.

Fuzzing hardware inputs. Besides receiving inputs from user space via system calls, an OS kernel also communicates with hardware devices. To detect driver bugs triggered by hardware inputs, several recent approaches [45, 51, 52] create a simulated device to generate and mutate hardware inputs to test drivers. They have found some security bugs caused by the bad data from DMA buffers. However, their code coverage is limited to generated test cases, causing that many real unsafe DMA accesses are missed. Besides, they cannot detect inconsistent DMA accesses, because they do not consider the synchronization of DMA buffers.

8.3 Symbolic Execution of OS Kernels

Some approaches [12, 14, 29, 46, 47] use symbolic execution to analyze OS kernels. However, symbolic execution is often time consuming in analyzing large-scale software, as it needs to explore numerous code paths and solve their path constraints. To reduce the overhead of solving path constraints, SADA first uses an efficient flow-sensitive analysis to detect all possible unsafe DMA accesses without validating code-path feasibility, and then uses an SMT solver to only validate the code paths of these possible unsafe DMA accesses.

8.4 Untrusted Hardware and Protection

A peripheral hardware device can be untrusted, and thus the attacker can use this device to attack the OS kernel. Some approaches [28, 53, 65] have proven that such untrusted hardware devices can be actually implemented to attack real-world systems. As a typical attack method from untrusted hardware, DMA attack can gain direct access to part or all of the system memory via untrusted DMA-enabled devices. To defend against DMA attacks, existing approaches [40–42, 44, 58] use IOMMU to limit the area of system memory that a DMA-enabled device can access. Even though IOMMU has been used to guarantee the memory addresses accessed by DMA are valid, DMA accesses can be still unsafe. SADA is designed to detect such unsafe DMA accesses in device drivers.

9 Conclusion

DMA is a frequently-used mechanism to improve hardware I/O performance, but DMA accesses can be unsafe and cause security problems. In this paper, we propose a static approach named SADA, to automatically and accurately detect unsafe DMA accesses in device drivers. SADA integrates three key techniques, including a field-based alias analysis to identify DMA accesses, a flow-sensitive and pattern-based analysis to check the safety of each DMA access, and a code-path validation method to drop false positives. In the Linux driver code, SADA finds 284 real unsafe DMA accesses, which can cause unexpected hardware behaviors or trigger security bugs (such as buffer overflow and invalid-pointer access).

In the future, we plan to apply SADA to other OSes (such as FreeBSD and NetBSD) to check their driver code. We also plan to add more patterns in checking the safety of DMA accesses, to find more unsafe DMA accesses that can trigger other kinds of security problems.

Acknowledgment

We thank our shepherd, Tuba Yavuz, and anonymous reviewers for their helpful advice on the paper. This work was mainly supported by the Natural Science Foundation of China under Project 62002195 and the China Postdoctoral Science Foundation under Project 2019T120093. Kangjie Lu was supported in part by the NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Shi-Min Hu is the corresponding author.

References

[1] ANDERSEN, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[2] BAI, J.-J., LAWALL, J., CHEN, Q.-L., AND HU, S.-M. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference* (2019), pp. 255–268.

[3] BAI, J.-J., LAWALL, J., AND HU, S.-M. Effective detection of sleep-in-atomic-context bugs in the Linux kernel. *ACM Transactions on Computer Systems (TOCS) 36*, 4 (2020), 1–30.

[4] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *Proceedings of the 1st European Conference on Computer Systems (EuroSys)* (2006), pp. 73–85.

[5] BIAN, P., LIANG, B., SHI, W., HUANG, J., AND CAI, Y. NAR-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 26th Symposium on the Foundations of Software Engineering (FSE)* (2018), pp. 411–422.

[6] HID: battery: do not do DMA from stack. https://github.com/torvalds/linux/commit/6c2794a2984f.

[7] RDMA: cxgb4: do not dma memory off of the stack. https://github.com/torvalds/linux/commit/3840c5b78803.

[8] 8139cp: Add dma_mapping_error checking. https://github.com/torvalds/linux/commit/cf3c4c03060b.

[9] tulip: Properly check dma mapping result. https://github.com/torvalds/linux/commit/c9bfbb31af7c.

[10] net: sxgbe: fix error handling in init_rx_ring(). https://github.com/torvalds/linux/commit/37c85c3498c5.

[11] usb: chipidea: udc: fix memory leak in _ep_nuke(). https://github.com/torvalds/linux/commit/7ca2cd291fd8.

[12] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th International Symposium on Operating Systems Design and Implementation (OSDI)* (2008), pp. 209–224.

[13] CALZAVARA, S., GRISHCHENKO, I., AND MAFFEI, M. HornDroid: practical and sound static analysis of Android applications by SMT solving. In *Proceedings of the 1st European Symposium on Security and Privacy* (2016), pp. 47–62.

[14] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), pp. 265–278.

[15] Clang: a LLVM-based compiler for C/C++ program. https://clang.llvm.org/.

[16] CLOC: count lines of code. https://cloc.sourceforge.net.

[17] CORBET, J., RUBINI, A., AND KROAH-HARTMAN, G. *Linux device drivers (3rd edition)*. O'Reilly, 2005.

[18] CORINA, J., MACHIRY, A., SALLS, C., SHOSHITAISHVILI, Y., HAO, S., KRUEGEL, C., AND VIGNA, G. DIFUZE: interface aware fuzzing for kernel drivers. In *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)* (2017), pp. 2123–2138.

[19] Clang static analyzer. clang-analyzer.llvm.org/.

[20] DEFREEZ, D., THAKUR, A. V., AND RUBIO-GONZÁLEZ, C. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 26th Symposium on the Foundations of Software Engineering (FSE)* (2018), pp. 423–433.

[21] DMA attack. https://en.wikipedia.org/wiki/DMA_attack.

[22] Detecting silent data corruptions and memory leaks using DMA debug API. https://events.static.linuxfound.org/sites/events/files/slides/Shuah_Khan_dma_map_error.pdf.

[23] DMA safety in buffers for Linux kernel device drivers. https://elinux.org/images/0/03/20181023-Wolfram-Sang-ELCE18-safe_dma_buffers.pdf.

[24] HAN, H., AND CHA, S. K. IMF: inferred model-based fuzzer. In *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)* (2017), pp. 2345–2358.

[25] HARDEKOPF, B., AND LIN, C. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 2011 International Symposium on Code Generation and Optimization (CGO)* (2011), pp. 289–298.

[26] JOHNSON, R., AND WAGNER, D. Finding User/Kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 1–22.

[27] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Tolerating hardware device failures in software. In *Proceedings of the 22nd International Symposium on Operating Systems Principles (SOSP)* (2009), pp. 59–72.

[28] KING, S. T., TUCEK, J., COZZIE, A., GRIER, C., JIANG, W., AND ZHOU, Y. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (2008).

[29] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference* (2010), pp. 1–14.

[30] LAWALL, J., AND MULLER, G. Coccinelle: 10 years of automated evolution in the Linux kernel. In *Proceedings of the 2018 USENIX Annual Technical Conference* (2018), pp. 601–614.

[31] LAWALL, J. L., BRUNEL, J., PALIX, N., HANSEN, R. R., STUART, H., AND MULLER, G. WYSIWIB: a declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)* (2009), pp. 43–52.

[32] LI, Z., AND ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on the Foundations of Software Engineering (FSE)* (2005), pp. 306–315.

[33] LLVM compiler infrastructure. https://llvm.org/.

[34] LU, K., AND HU, H. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th International Conference on Computer and Communications Security (CCS)* (2019), pp. 1867–1881.

[35] LU, K., PAKKI, A., AND WU, Q. Automatically identifying security checks for detecting kernel semantic bugs. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)* (2019), pp. 3–25.

[36] LU, K., PAKKI, A., AND WU, Q. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium* (2019), pp. 1769–1786.

[37] LU, K., WANG, P.-F., LI, G., AND ZHOU, X. Untrusted hardware causes double-fetch problems in the I/O memory. *Journal of Computer Science and Technology (JCST) 33*, 3 (2018), 587–602.

[38] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: a soundy analysis for Linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 1007–1024.

[39] MAO, J., CHEN, Y., XIAO, Q., AND SHI, Y. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016), pp. 531–544.

[40] MARKETTOS, T., ROTHWELL, C., GUTSTEIN, B. F., PEARCE, A., NEUMANN, P. G., MOORE, S., AND WATSON, R. Thunderclap: exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[41] MARKUZE, A., MORRISON, A., AND TSAFRIR, D. True IOMMU protection from DMA attacks: when copy is faster than zero copy. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016), pp. 249–262.

[42] MARKUZE, A., SMOLYAR, I., MORRISON, A., AND TSAFRIR, D. DAMN: overhead-free IOMMU protection for networking. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2018), pp. 301–315.

[43] PAILOOR, S., ADAY, A., AND JANA, S. MoonShine: optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 729–743.

[44] PELEG, O., MORRISON, A., SEREBRIN, B., AND TSAFRIR, D. Utilizing the IOMMU scalably. In *Proceedings of the 2015 USENIX Annual Technical Conference* (2015), pp. 549–562.

[45] PENG, H., AND PAYER, M. USBFuzz: a framework for fuzzing USB drivers by device emulation. In *Proceedings of the 29th USENIX Security Symposium* (2020), pp. 2559–2575.

[46] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: correctness checking for real code. In *Proceedings of the 24th USENIX Security Symposium* (2015), pp. 49–64.

[47] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: testing drivers without devices. In *Proceedings of the 10th International Symposium on Operating Systems Design and Implementation (OSDI)* (2012), pp. 279–292.

[48] SAHA, S., LOZI, J., THOMAS, G., LAWALL, J. L., AND MULLER, G. Hector: detecting resource-release omission faults in error-handling code for systems software.

In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)* (2013), pp. 1–12.

[49] SCHWARZ, M., GRUSS, D., LIPP, M., MAURICE, C., SCHUSTER, T., FOGH, A., AND MANGARD, S. Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS)* (2018), pp. 587–600.

[50] SHERMAN, E., GARVIN, B. J., AND DWYER, M. B. Deciding type-based partial-order constraints for path-sensitive analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM) 24*, 3 (2015), 1–33.

[51] SONG, D., HETZELT, F., DAS, D., SPENSKY, C., NA, Y., VOLCKAERT, S., VIGNA, G., KRUEGEL, C., SEIFERT, J.-P., AND FRANZ, M. Periscope: an effective probing and fuzzing framework for the hardware-os boundary. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[52] SONG, D., HETZELT, F., KIM, J., KANG, B. B., SEIFERT, J.-P., AND FRANZ, M. Agamotto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the 29th USENIX Security Symposium* (2020), pp. 2541–2557.

[53] STURTON, C., HICKS, M., WAGNER, D., AND KING, S. T. Defeating UCI: building stealthy and malicious hardware. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy* (2011), pp. 64–77.

[54] SUI, Y., YE, D., AND XUE, J. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering (TSE) 40*, 2 (2014), 107–122.

[55] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS) 24*, 4 (2006), 333–360.

[56] Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[57] TALEBI, S. M. S., TAVAKOLI, H., ZHANG, H., ZHANG, Z., SANI, A. A., AND QIAN, Z. Charm: facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 291–307.

[58] TIAN, K., ZHANG, Y., KANG, L., ZHAO, Y., AND DONG, Y. coIOMMU: a virtual IOMMU with cooperative DMA buffer tracking for efficient memory management in direct I/O. In *Proceedings of the 2020 USENIX Annual Technical Conference* (2020), pp. 479–492.

[59] VOJDANI, V., APINIS, K., RÕTOV, V., SEIDL, H., VENE, V., AND VOGLER, R. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)* (2016), pp. 391–402.

[60] WANG, P., KRINKE, J., LU, K., LI, G., AND DODIER-LAZARO, S. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 1–16.

[61] WANG, P., LU, K., LI, G., AND ZHOU, X. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience 30*, 6 (2018), e4345.

[62] WANG, P., LU, K., LI, G., AND ZHOU, X. DFTracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science 13*, 2 (2019), 247–263.

[63] WANG, W., LU, K., AND YEW, P.-C. Check it again: detecting lacking-recheck bugs in OS kernels. In *Proceedings of the 25th International Conference on Computer and Communications Security (CCS)* (2018), pp. 1899–1913.

[64] XU, M., QIAN, C., LU, K., BACKES, M., AND KIM, T. Precise and scalable detection of double-fetch bugs in OS kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy* (2018), pp. 661–678.

[65] YANG, K., HICKS, M., DONG, Q., AUSTIN, T., AND SYLVESTER, D. A2: analog malicious hardware. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016), pp. 18–37.

[66] Z3: an effective theorem prover from Microsoft Research. https://github.com/Z3Prover/z3.

[67] ZHANG, T., SHEN, W., LEE, D., JUNG, C., AZAB, A. M., AND WANG, R. PeX: a permission check analysis framework for Linux kernel. In *Proceedings of the 28th USENIX Security Symposium* (2019), pp. 1205–1220.