Journal of Information Processing Vol.24 No.3 573–582 (May 2016)

[DOI: 10.2197/ipsjjip.24.573]

Regular Paper

A Case for Static Analysis of Linux to Find Faults in Interrupt Request Handlers

Takeshi Yoshimura[1,a)] Kenji Kono[1,b)]

Received: July 31, 2015, Accepted: November 13, 2015

Abstract: Bugs in operating system kernels threaten system reliability and availability. Static analysis of device drivers is one of the most useful methods to find and fix bugs in operating systems. Unfortunately, existing tools focus on bug patterns that come from developers' ad hoc beliefs and experiences, although the developers have a chance to utilize many past bug reports. The objective of this paper is to uncover particular types of real bugs in a widely used operating system. Specifically, this paper presents a case for finding six real bugs in Linux when obtaining 160 bug reports about interrupt request line (IRQ) handlers in past Linux. The 160 bug reports enable us to recognize nine patterns of mishandling IRQ handlers, and our analyzer, which is based on the recognized patterns, successfully detects the uncovered bugs.

ad hoc: 直观的

Keywords: operating system, dependability, static analysis, symbolic execution

1. Introduction

Static analysis is one of the most useful methods for finding bugs in software. Numerous tools and techniques for static analysis have been proposed to debug large-scale C code. A good example of large-scale C code is an operating system such as Linux and Windows because these code bases are huge and complicated. In fact, static analysis is widely used to find bugs in Linux and Windows, especially in their device drivers [1], [2], [3], [4], [5], [6], [7], [8], [9].

The key concept of static analysis to find bugs is identifying typical bug patterns in code. For example, the Clang Static Analyzer can search code locations for potential NULL pointer dereferences. However, typical bugs in complex software like Linux are not only such generic C programming mistakes but also other domain-specific issues. A field study of Linux bugs in file systems [10] reveals that file system-specific bugs are dominant. Additionally, the sheer number of the previous work for finding unknown bug patterns ([1], [3], [5], [6], [9], [11], [12], [13], etc.) suggests that new bug patterns will continue to show up in the future. In other words, static analysis assumes developers are continuously making efforts to keep up with new typical bug patterns.

Nevertheless, one of the biggest obstacles to finding bugs is to recognize what to check as mentioned by Engler et al. [9]. Much work tackles the issue by using static analysis with pattern inference engines [9], [14], dynamic state tracking [13], and tools for exploiting developer's knowledge and experiences [12], for example. Unfortunately, these efforts implicitly focus on bugs that are often based on a developer's ad hoc beliefs and experiences, rather than actual typical bugs reported in the past.

Our prior work [15] shows preliminary results of using natural language processing for static analysis developments. Specifically, the work reports nine bug patterns from a patch group and two bug types they found in Linux PCI device drivers. However, their focus is to extract a group of similar patches from a large number of patches, and thus, it does not fix bugs. Besides, the work lacks details of the bug patterns they identify and implementation details of static analysis. This paper presents our experiences of each process to uncover real bugs.

The objective of this paper is to uncover particular types of real bugs in a widely used operating system. Specifically, this paper presents a case for making use of 160 past bug reports to develop static analysis to find bugs in a commodity operating system. The used reports are derived from the results of natural language processing with more than 370,000 patch documents [15]. With such a methodology for "big-data processing", developers can directly recognize real problems and what to check. However, there remains a gap between extracting typical bugs and obtaining static checkers. Developers still need to understand system behaviors to be validated, select an appropriate static analysis framework, and implement checkers along with the framework characteristics.

The major contribution of this work is to find and fix six real bugs in Linux device drivers. At the time of this paper, five fixes are accepted by Linux maintainers and landed on the upstream kernel. All the bugs our static analysis found have existed for three to ten years in Linux, although it has been one of the most widely used pieces of software in the world for decades. The 160 bug reports enable us to recognize nine patterns of misusing IRQ handlers, and our analyzer, which is based on the recognized patterns, successfully detects the uncovered bugs. The result implies that utilizing software repositories for static analysis is promising to enhance future software quality by detecting bugs that many

[1] Keio University, Yokohama, Kanagawa 223–8522, Japan
[a)] yos@sslab.ics.keio.ac.jp
[b)] kono@ics.keio.ac.jp

developers overlook.

The process of detecting real bugs in this work starts with a study of 160 bugs about freeing IRQ handlers with relevant APIs in Linux. Our investigation reveals four typical misuses of in-kernel APIs. The point of the studied bugs is that they potentially cause serious failures in operating systems, kernel deadlocks, or unexpected interleaving of interrupt handlers, which all techniques find hard to avoid and recover from without system rebooting. Our observation also shows that the failures happen under transient conditions such as particular error occurrences and environments, which other bug-finding techniques are hard to cover.

The bug reports in the past informed us of three challenges to finding the bug patterns with static analysis. First, the static analyzer cannot track down the whole state transitions of IRQ handling on complex, dynamic execution-flows due to external event handling. Second, various kinds of driver classes can cause IRQ handler misuses although they have different semantics for their event callbacks in Linux. Third, the static analyzer needs to identify the time when drivers should free a resource in the case of checking resource-release omissions within the lifetime of device drivers.

On the other hand, past bugs simultaneously give us the idea that IRQ handling often happens among event callbacks for drivers. Based on this idea, our solution to the challenges is to inject analysis-dedicated code that emulates the typical execution-flows of device drivers in a .c file with which static analysis runs. The code is just a function that invokes event callbacks in the typical order. Although there can be many, complicated flows of these driver invocations, we expect that developers can easily develop the emulation function by utilizing both symbolic execution characteristics and abundant C language expressions. Besides, the emulation function enables developers to validate complex state transitions involving external events in the manner of simple inter-procedural analysis.

Our bug-finding tool implementation uses the framework of the static analyzer in Clang 3.7. It consists of the main static analysis engine and code injector to assist static checking. The code injector automates the process of writing emulation functions and scale to 2287 device drivers in Linux 4.1-rc1. In this work, we perform static analysis of 8 driver classes in Linux: generic (platform drivers in Linux), Peripheral Component Interconnect (PCI), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), network operations (open, close), control and measurement device interface (Comedi), and PCMCIA devices.

In the experiment, the tool detects 598 device drivers that manage IRQ handling in Linux. The problem of false positives in static analysis is out of the scope of this work, although more than a hundred happen. To find bugs more practically, we obviously need to reduce the number of false positives in future work as existing work already has [3].

The rest of this paper is organized as follows. Section 2 gives a detailed overview of target bugs and discusses the design choice of bug-finding tools that are specialized to the obtained bug pattern. Sections 3 and 4 describe our strategy for developing a specialized static analysis. Section 5 shows our experiments on the latest Linux to show the capability of our tool to find bugs. Section 6 explains our related work, and Section 7 concludes this paper.

2. Analysis of Real-world Bugs for IRQ handling

In this section, we show challenges to detecting particular bugs in the real world with static analysis even if the patterns are well-known pairwise misuses. Specifically, we show 160 mistakes of managing IRQ handlers in Linux as a case we solve in this work. We observed most of the bugs were mistakes of releasing IRQ handlers in device drivers. In particular, we frequently observed misuses of request_irq() and free_irq().

2.1 API Semantics and the Programming Model

Before discussing the investigated bugs, we briefly describe API specification for ease of understanding the issue discussed in this section. API specifications we validate are in **Fig. 1**. Request_irq() and free_irq() are in-kernel APIs for the registration of IRQ handler in Linux. They require various arguments, including an interrupt request number (irq), a flag of interrupt types, a function pointer for the interrupt handler corresponding to the irq, and an extra variable (dev_id). Free_irq() is an API for releasing a requested irq by specifying the irq and dev_id. Linux uses dev_id to validate requesting and releasing an irq shared among multiple drivers.

Also, we need to consider the programming model of Linux device drivers to understand the problem we are trying to solve. In particular, Linux device drivers often offer event-driven programming. The Linux kernel core dynamically invokes driver callbacks that the driver initialization routine registered for each external event such as physical device probe, removal, and power management.

Figure 2 shows typical API usages in device drivers to be checked. Drivers often store driver-specific states like IRQ numbers to given callback arguments (struct X *x in the example). The usage of the APIs is similar to that of other typical pairwise APIs (e.g., malloc/free, lock/unlock). However, there are subtle but significant differences for checker implementation in practice. For example, when checking a shared IRQ, we need to validate the consistency of two arguments unlike malloc/free, lock/unlock. Another primary difference is that drivers know the IRQ number before calling request_irq(). This means we do not need to validate accidental free_irq() on request-failed IRQs while failed

```
//return 0 on success (negative on failures)
int request_irq(
  unsigned int irq,       // Interrupt line
  irq_handler_t handler,  // interrupt handler func.
  unsigned long irqflags, // Interrupt type flags
  const char *  devname,  // Device name
  void * dev_id);         // Device identity

void free_irq(
  unsigned int irq,       // Interrupt line
  void * dev_id);         // Device identity
```

Fig. 1 Two checked API declarations and comments for each argument (declared in include/linux/interrupt.h).

```
int x_probe(struct X * x) {
    /* various resource initializations */
    ...
    if (request_irq(x->irq, x_isr, ..., x))
        goto err1;
    x->some_src = some_src_alloc();
    if (!x->some_src)
        goto err2;
    return 0;
err2:
// request_irq() must be revoked on failures
    free_irq(x->irq, x);
err1:
    /* release resources */
    return err;
}

void x_remove(struct X * x) {
    /* various release resources */
    ...
    free_irq(x->irq, x);
    ...
}
```

Fig. 2 Typical API usages.

Table 1 Investigated bugs. Natural language processing (NLP) in our prior work [15] extracts a cluster for around 5000 patches whose keywords contain 'irq'. NLP shows us what keywords each patch explanation have. We further manually investigated 331 patches from them by selecting patches whose keywords also contain 'free'. The table lists 160 bugs we identified in the patches.

Bug type	Description	Num.
Argument	free_irq() with inconsistent dev_id	41
Argument	free_irq() with an invalid irq number	25
Leak	missing free_irq() at driver initialization errors	25
Leak	missing free_irq() at driver unloading	13
Leak	missing free_irq() before device is suspended	6
DoubleFree	double free_irq()	9
Order	releasing other src before free_irq()	7
Order	releasing pages with interrupt disabled	7
Order	freeing shraed irq with interrupt disabled	5
Other		22
Total		160

malloc() returns NULL pointer that free() ignores. In practice, catching request_irq() failures is more error-prone than expected, especially when requesting multiple IRQs as shown in an example in Section 2.2.

2.2 Bug Patterns

Table 1 shows our observation results of bugs extracted in our prior work [15] [*1]. We define four bug types: `Argument`, `Leak`, `DoubleFree`, `Order`. Each type has some of the minor divisions. Inconsistent arguments (`Argument`) are the major category of bug patterns. Missing free_irq() is the second largest category in our observation (`Leak`), although `Argument` bugs consequently cause the same effect as `Leak` bugs. Like for general bugs, double-frees (`DoubleFree`) and order violations of releasing resources (`Order`) were observed. `Order` types had seven cases that were not for interrupt handler registrations. Table 1 shows all but `Order` type were the imbalance of API usages.

[*1] The mining accumulates patches that have topics for 'irq' and 'free' by using natural language processing. Thus, the result includes not only bugs for free_irq(), although most of the bugs were relevant to it. The prior paper [15] describes details of the mining method and results with the preliminary result of our static checking.

drivers/misc/max8997-muic.c, Linux 3.4-rc3, commit 3241d56edda5

```
max8997_muic_probe(...) {
    for (i = 0; i < ARRAY_SIZE(muic_irqs); i++) {
        ret = request_threaded_irq(...);
        if (ret) {
            ...
-           for (i = i - 1; i >= 0; i--)
-               free_irq(muic_irq->irq, info);
            goto err_irq;
...
err_irq:
+   while (--i >= 0)
+       free_irq(pdata->irq_base
+           + muic_irqs[i].irq, info);
err_pdata:
    kfree(info);
```

Fig. 3 Example of Leak bug in an error path.

drivers/tty/serial/samsung.c, Linux 3.9-rc3, commit b6ad29355560

```
s3c64xx_serial_startup(struct uart_port *port) {
    ...
    ret = request_irq(port->irq,..., ourport);
    ...
}
...
s3c24xx_serial_shutdown(struct uart_port *port) {
    if (s3c24xx_serial_has_interrupt_mask(port)) {
+       free_irq(port->irq, ourport);
        wr_regl(port, S3C64XX_UINTP, 0xf);
...
```

Fig. 4 Example of Leak bug depending on user inputs.

Figure 3 shows an `Argument` bug in the unified diff format of its fix. One of the typical mistakes of freeing IRQs occurs during the failure paths like that in the example. Typical device drivers initialize their IRQ handlers as well as other resources. However, the resource initializations might fail. It means drivers have to revoke all the acquired resources as if the system did not load the driver in such cases. Before fixing the bug in Fig. 3, the driver frees only an IRQ after failing to request an IRQ although she intended to free all the allocated IRQs. These kind of bugs in failure paths are difficult to find by testing in debugging environments. In this case, requesting IRQs rarely fails; we may encounter such rare cases when we load a particular device driver that (un)intentionally uses the same IRQ number.

Figure 4 is an example of a `Leak` bug. The driver requests an IRQ when it opens a serial port and frees the IRQ when it shutdowns the serial port. We can check the example bug by loading and unloading the driver with specific models of Samsung system-on-chips that make s3c24xx_serial_has_interrupt_mask(port) true. However, not all the maintainers have the specific models, and the models might be rare or old ones in the future. Also, there are too many device drivers that handle IRQs as described in the next section. Thus, such runtime testing by using physical devices is time-consuming and not cost-effective for checking a large number of drivers.

Most bugs in Table 1 potentially cause serious consequences in systems although they are difficult to test by running systems. For example, no other device driver can use an IRQ number until the system shutdowns as the consequences of `Leak` bugs like in

Table 2 Callbacks for registering and releasing IRQs.

Callers of request_irq()		Callers of free_irq()	
struct name::member name	Num.	struct name::member name	Num.
platform_driver::probe	398	pci_driver::remove	253
pci_driver::probe	329	platform_driver::probe	240
i2c_driver::probe	204	pci_driver::probe	210
net_device_ops::ndo_open	121	platform_driver::remove	175
spi_driver::probe	72	i2c_driver::probe	145
platform_driver::remove	62	net_device_ops::ndo_stop	116
pci_driver::resume	42	i2c_driver::remove	108
work_struct::func	38	comedi_driver::detach	102
pcmcia_driver::probe	34	net_device_ops::ndo_open	67
comedi_driver::auto_attach	31	spi_driver::probe	56

the example in Fig. 4. In the case of `Argument` bugs, we may also unintentionally release IRQ handlers in other running device drivers. In other words, we do not release an IRQ handler in `Argument` bugs like the example in Fig. 3. `DoubleFree` bugs cause either just redundant executions or missing free_irq() calls, depending on the developers' intention. The manifestation of `Order` bugs depend on the timing of device interrupts, context switches, and concurrent executions. For example, an interrupt handler may access invalid heap memory, and memory-mapped I/O (MMIO) if the driver releases the MMIO and memory earlier than IRQ.

2.3 Candidates of Fault Sites

The examples in Figs. 3 and 4 suggests mistakes on IRQ releases happen on device drivers, i.e., loadable kernel modules in Linux kernel. An existing tool to detect intra-procedural resource-release omissions [3] shows the cases for initialization failures like in Fig. 3. However, the example also implies there can be more difficult cases involving user operations like in Fig. 4.

To confirm the validity of the implication, we analyzed at which event device drivers call request_irq() and free_irq() in Linux 4.1-rc1. The analysis first detects the root functions whose call graph includes at least one of the calls of two API and their family such as request_threaded_irq(). Then, it searches for an event callback in a device-specific struct (e.g., pci_driver). Finally, we join the two analysis results so as to obtain event callbacks that call the two API families. We describe the details of extracting event callbacks in Section 4.2.

Table 2 shows a part of the analysis results. The biggest users of the IRQ handler APIs were generic device drivers (function pointers in struct platform_driver) and PCI drivers (function pointers in struct pci_driver). As the example indicates, drivers call request_irq() at driver constructions such as device probes, opens, and resumes. Free_irq() occurs at driver destructions such as device removal and failure paths in device probes. The result shows that checkers should consider not only API pairwise confined within an event callback like in Fig. 3, but also API pairwise crossing event callbacks like in Fig. 4. Also, it shows there are many kinds of device drivers that request IRQs that potentially cause bugs described in Section 2.2.

2.4 Summary and Discussion

Our observation of IRQ bugs in this section shows that there are many imbalances of request_irq() and free_irq() in the real world. This indicates that developing static checkers specific to validating the balance of the pairwise like Ref. [3] can effectively reduce debugging efforts during device driver developments. This is because the bug characteristics offer high coverages that dynamic testing cannot achieve.

However, Table 2 and past examples show that we need to track API pairwise crossing event callbacks. We do not focus on `Order` bugs in this work because they are less frequent than API imbalances, and such timing-dependent bugs should be covered by fuzzing tests like Ref. [16].

3. Finding mistakes of IRQ handling

In this work, we use symbolic execution for our inter-procedural, path-sensitive static analysis to find IRQ bugs on the basis of our observation in Section 2. One of the biggest advantages of symbolic execution is that it can achieve high coverages from normal paths to exceptional, rarely executed paths even without running a target system [2]. Table 1 shows 25 bugs appeared on rarely-executed paths such as error paths in driver initializations. Besides, we observed many `Argument` bugs on failure paths like the example in Fig. 3. Symbolic executions also enable us to check such inconsistency of symbolic (or concrete, if possible) values for corresponding arguments.

3.1 Workflow

Figure 5 shows our abstract workflow to detect IRQ bugs. First, our tool generates checked code from the original driver code and specifications of driver lifecycle (i.e., event callback execution-flows) given by users. Then, the execution engine runs on each translation unit (e.g., single .c file and included .h files) until it completes analyzing all the translation units. During the execution, our analyzer simply checks the state of each IRQ handler to validate the balance of API usages. After all the analyses have finished, our bug-finding tool summarizes bug reports in HTML formats.

We do not analyze driver complete binaries because our observation indicates that uses of pairwise API like IRQ handling mostly appeared in the same .c file. In other words, our tool ensures that device drivers confine their unit of system rules [1] within a translation unit. For example, our tool alerts us of a potential bug if a pair of request_irq() and free_irq() appears in different .c files.

3.2 IRQ State Tracking

Figure 6 shows our simplified version of an IRQ state transitions that our tool tracks and checks. At the beginning of ana-

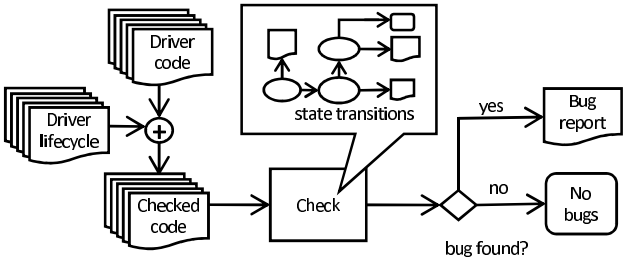

Fig. 5 Analysis workflow.

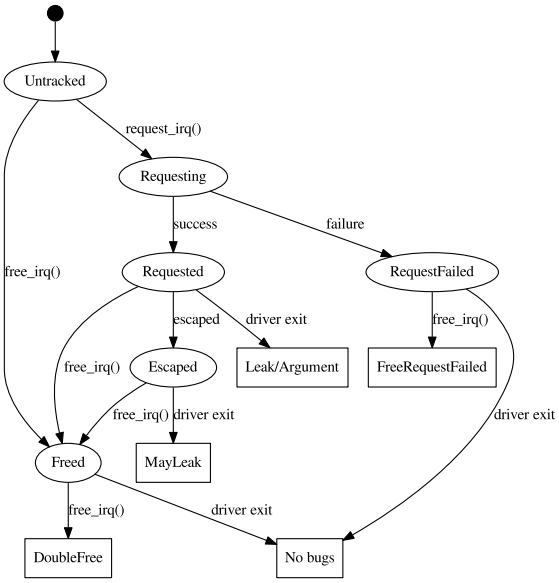

Fig. 6 IRQ state transitions. Circles are the main states of IRQs, while rectangles are the consequence of state transitions.

lyzing a driver, the checker assumes the state is not tracked (*Untracked*). When the analyzed driver calls request_irq(), the analyzer stores the symbolic value for the arguments and moves the state to *Requesting*. Then, it bifurcates the state into *Requested* and *RequestFailed* for the succeeded and failed paths, respectively. *FreeRequestFailed* represents erroneous cases where drivers free non-existing IRQs specialized for request-failed (i.e., not-allocated) ones. By the bifurcation with symbolic executions, our checker can independently check two possible state transitions after request_irq() returns. Thus, after *RequestFailed*, our analyzer checks if free_irq() is called or not. Finally, it moves a *Requested* state to *Freed* when the driver calls free_irq() with the consistent symbolic values of the arguments. In current Linux, free_irq() never fails, and thus, there are no bifurcations after free_irq(). The report of a *Leak* bug appears when the driver does not call free_irq() for a *Requested* state at the end of the analysis. We keep the value pairs even after freed IRQs so that we can report *DoubleFree* bugs if a driver frees an IRQ twice without requesting it again.

The analysis focused on a translation unit potentially overlooks bugs when an analyzed driver passes tracked states via pointers to external functions outside of a translation unit. For an example of IRQ numbers, drivers often store them inside a single struct variable instantiated for each driver (see the first argument of request_irq() in Fig. 4). When the driver passes the pointer to the variable to an external function, the analyzer cannot detect the modification of tracked states by the function. As other checkers do in the Clang Static Analyzer, we mitigate the issue by introducing an *Escaped* state. The *Escaped* state may confuse users by emitting false reports, but it enables users to prioritize inspecting reports with fewer false negatives. Thus, users can start their report inspections by using more doubtful reports such as *FreeRequestFailed*, *Double Free*, and *Leak/Argument* before less doubtful ones like *MayLeak*. Thus, the *Escaped* state is important for users to reduce manual efforts to find bugs in large-scale, complex source code such as operating systems.

When the driver passes the pointer to the variable representing a *Requested* state to an external function, the analyzer turns the state to *Escaped*. We do not change states other than *Requested* because we rarely observe external functions that call free_irq() i.e., *Double Free* and *FreeRequestFailed* rarely happen in external functions. The analyzer treats *Escaped* states like *Requested* states except for tracked values. For *Escaped* states, it tracks symbolic values of the address that stores the values that the analyzer tracked for the *Requested* state beforehand. When the analyzer detects the symbolic value of the address for arguments of free_irq(), it moves the *Escaped* state to *Freed*. Note that the analyzer reports all the analysis results that went through *Escaped* states (e.g., *MayLeak*) in order to avoid false negatives.

However, we cannot always track *Escaped* states. For example, we generate *Leak* reports when the address for the tracked state is potentially modified, although they might be false ones. This limitation affects the strategy of detecting Argument bugs. In Fig. 6, free_irq() to an *Untracked* state transits the state to a Freed state. The analyzer could report *Argument* bugs when there was no consistent pair of arguments of free_irq() in stored arguments. In that case, however, escaped states cause false reports. Thus, we alternatively detect *Argument* bugs as *Leak* bugs.

3.3 Execution-flow Emulation

The previous section described how we manage and check state transitions inside a static execution-flow. However, execution-flows of Linux device drivers are not always static because of the programming model of Linux device drivers.

To forge events at symbolic execution time, we inject an emulating function, which simply calls registered callbacks for events in a typical order. We do not modify or re-use existing execution engines so that we can avoid the complexity increases of IRQ state tracking. Existing symbolic execution engines for C language can already emulate execution-flows that mainly appear in every function definition because of the nature of C language. Thus, we can emulate execution-flows at static analysis by adding a function that invokes callbacks in a typical event order.

To write the emulating function, we first need to identify the typical event order at runtime. We describe it with an example of PCI device drivers in this section. A typical PCI driver execution-flow is **Fig. 7**.

Considering that we focus on the exeuction-flow in Fig. 7, the emulation code should first call probing callback in PCI device drivers immediately after the symbolic execution starts. Then, the symbolic execution engine bifurcates the checker execution into two because PCI device drivers in Linux sometimes fail to initial-

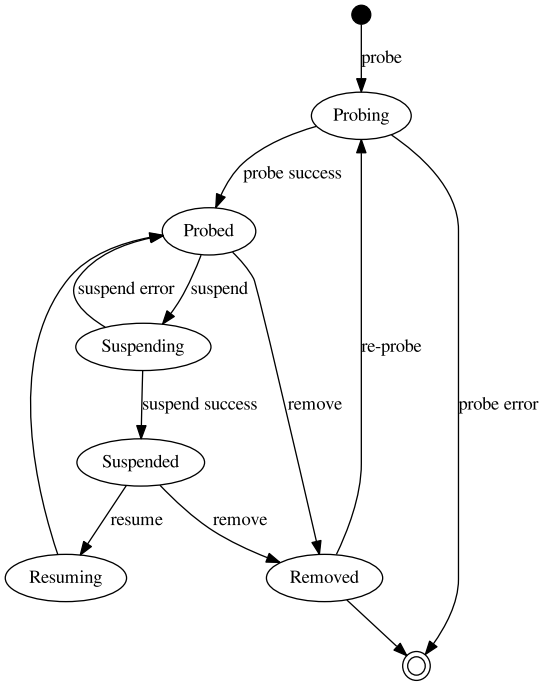

Fig. 7 PCI driver execution-flow.

ize their resources in probing functions. After probing devices, the driver might have to handle suspending or physical removal events. A suspending event can fail, so the execution engine bifurcates the checker execution as well as for probing events. On the other hand, removal and resuming events do not fail in Linux. Note that removal events can always happen physically except that Linux guarantees the event atomicity like resuming; device drivers have to handle even when users remove suspended devices or re-probe removed devices.

Figure 8 is the simplified version of injected code. We inject the code simply by appending a code fragment like in Fig. 8 to existing .c files. Each step of the driver invocation corresponds to driver callbacks (e.g., for a device probe, the injected code calls a struct pci_driver::probe()). The Linux documentation describes that the kernel core invokes PCI drivers by calling function pointers in struct pci_driver registered at the initialization of a device driver. We detect the defined callbacks by traversing abstract syntax tree to find initialization of function pointers in struct pci_driver as well as static analysis in Section 2.3. We manually write the specification template like Fig. 8 and automatically replace x_* function with the functions obtained in the static analysis.

Our code injection is to utilize symbolic execution properties. The injected code uses an external dummy function (random() in the figure) to bifurcate the execution into more than two. Also, symbolic execution engine stores a single generic driver state (pdev in the figure), and we can share the state among callback invocations by passing it as a function argument. In the Linux kernel, most of the drivers store dynamically allocated driver-specific states like IRQ numbers to the generic driver states. Thus, the injected function has parameters including driver state to create symbolic values for IRQ numbers in any invoked function. If the driver modifies the symbolic value for irq or dev_id after calling request_irq(), our tool can detect the imbalance of IRQ handling

```
#ifdef __clang_analyzer__
extern int random();

static void TestPCIDriver(struct pci_dev *pdev,
                const struct pci_device_id *id) {
        int loop = 0;
        enum PCI_STATE state;
reprobe:
        if (x_probe(pdev, id)) {
                return;
        }
        state = PCI_STATE_PROBED;

normal_operation:
        switch(random() % 4) {
        case PCI_EVENT_PM:
                x_power(pdev, &state);
                break;
        case PCI_EVENT_REMOVE:
                x_remove(pdev);
                state = PCI_STATE_REMOVED;
                if (random() % 2 && loop++ < 10) {
                        goto reprobe;
                }
                break;
        default:
            /* do nothing */
        }
        if (random() % 2 && loop++ < 10
               && state == PCI_STATE_PROBED) {
                goto normal_operation;
        }

        x_shutdown(pdev);
}
#endif
```

Fig. 8 Example of injected code. x_probe, x_remove, etc. are calling pci_driver::probe(), remove(), etc. x_power is a function to emulate the state transition of suspends, hibernations, etc.

APIs.

Additionally, our code injection is to utilize rich C expressions to define driver execution-flows. For example, we use a loop counter to conduct a bounded number of re-probing. We also use a local variable to maintain the current state of the PCI driver. The extra requirements for learning domain-specific languages to define driver execution-flow are not necessary. Thus, we can easily extend the emulating function implementation to other driver classes than ones that we check in this work.

4. Implementation

We implement our bug-finding tool as a plugin of Clang 3.7. The tool consists of two components: IRQ state tracker and code injector for emulating typical driver execution-flows in static analysis. The Clang Static Analyzer hooks compiler invocations and runs the analysis with the code and the same compiler options. We analyzed the Linux kernel with all options enabled (i.e., allyesconfig).

The IRQ state tracker described in Section 4.1 utilizes rich compiler-level information such as ASTs, call graphs, symbolic value information, and so on. The implementation consists of 1374 lines of C++11 and 264 lines of Python script. Clang provides us the framework for customized static analysis including the symbolic execution engine.

4.1 IRQ State Tracker

During symbolic executions, our state tracker hooks the calls of request_irq() and free_irq() to track IRQ state transitions. However, to analyze only specified driver execution-flows, it ignores these two call expressions when the root of the traversed call sequence is not a specified entry function ("Test*" in our prototype).

At request_irq(), our state tracker makes new concrete value and symbolic value for both succeeded and failed return code of the API. Specifically, the succeeded value is zero (constant), and the failed value is a symbolic value for lower than zero. Then, our state tracker creates two execution contexts that add the new value constraints to the current symbolic execution state to emulate both succeeded and failure path of request_irq(). Request_irq() and free_irq() are external functions for checked drivers, and thus, it is necessary for our static analysis

When the execution hits an expression that calls APIs request_irq() or free_irq(), our state tracker extracts symbolic or concrete values for two arguments of the APIs: irq and dev_id. In the Linux kernel, paired values of the irq and dev_id represent IRQ identification. Thus, we use symbolic or concrete values of them to identify if the released pair was already requested by the driver, for example. If our state tracker confirms an IRQ state remains *Requested* at the end of the entry function, it generates a Leak bug report.

Escaped states can appear on the expression of function calls with any pointers. After detecting the expression, we identify type information of a pointed variable if the pointer passing potentially modifies values of irq and dev_id. If the escaped pointer can reach either irq or dev_id via struct member access or others, we mark the IRQ state *Escaped*. We assume no buffer overruns that modify irq and dev_id because buffer overruns should be detected in other bug-finding tools. Other statements such as temporarily copying to global variables can cause *Escaped* states, but we do not handle such cases currently because we did not observe many and can recognize them when we manually check bug reports.

4.2 Code Injector

Manual implementations in Fig. 8 for each driver costs too much in terms of engineering. Thus, we implment a code injector that automates the process of identifying the registered callbacks and generating injected functions.

Linux device drivers often register callbacks stored in constant variables like in **Fig. 9**. We recursively parse all the initialization statements to pick up declared function names passed as function pointer. Specifically, we look up the left-hand side whose type is a function pointer. If so, we record the identifier of the left-hand side (with the struct type name), and the right-hand side. In the example, we first obtain 'pci_driver', 'probe', and 'e1000_probe'. The callback interface allows us to modify the callback functions, but we ignore such cases because drivers often do not change them dynamically.

After the callback identification, we inject code generated from a template we wrote. We carefully wrote the template so that we can emulate possible event orders as described in Section 3.3. We

drivers/net/ethernet/intel/e1000/e1000_main.c

```
static struct pci_driver e1000_driver = {
        .name     = e1000_driver_name,
        .id_table = e1000_pci_tbl,
        .probe    = e1000_probe,
        .remove   = e1000_remove,
#ifdef CONFIG_PM
        /* Power Management Hooks */
        .suspend  = e1000_suspend,
        .resume   = e1000_resume,
#endif
        .shutdown = e1000_shutdown,
        .err_handler = &e1000_err_handler
};
```

Fig. 9 An example of callbacks we collect.

refered to Linux documents and driver implementations to learn the possible event orders. For PCI drivers, the example code in Fig. 8 is extended to track physical device errors that typical PCI protocol defines (err_handler in Fig. 9). On the basis of our observation in Section 2, we focus on 8 driver classes: generic (platform drivers in Linux), Peripheral Component Interconnect (PCI), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), network operations (open, close), control and measurement device interface (Comedi), and PCMCIA devices. Finally, we wrote 588 lines of C code for the template.

5. Experiments

We inject emulation code into 2287 drivers in the Linux 4.1-rc1 and finally checked 598 drivers that managed IRQ handlers. Our tool generated 60 bug reports (i.e., *Leak/Arguments*, *Double Free*, and *FreeRequestFailed*), 177 *MayLeak* reports, and 294 *Escaped* reports. The 60 bug reports are more likely to contain bugs because the analyzer completely tracked symbolic values related to IRQ API uses. *MayLeak* reports may contain *Leak* bugs in which the analyzer detects no free_irq() calls with a requested IRQ, although it can track state transitions to *MayLeak* in Fig. 6. *Escaped* reports may contain bugs that the analyzer cannot track (*MayLeak* reports are excluded), but are more likely to be false positives than other report types. Our experiment runs on a single thread with Intel Xeon X5650 2.67 GHz and 15 Gbytes RAM on HP ProLiant DL360 G7. Our static analysis required 13.2 hours for generating emulation code of driver lifecycles and 7.4 hours for checking state transitions. Our manual investigation started with the more suspicious of the 60 bug reports and then moved on to the less suspicious ones and found six cases of real bugs within two weeks. When we found suspicious code, we wrote and sent a patch to Linux maintainers in order to validate our results. Five out of six patches were accepted and will be merged into the upstream version of Linux. At the time of writing, one patch we wrote had not been accepted because a developer responding to our report pointed out problems of our patch in code other than fixed IRQ handling. Because we cannot judge if or not he determines the fixed code to be a bug, **Table 3** only shows "Not yet." In this section, we report details of bugs around IRQ handling we found.

Table 3 overviews the result of our checking. five out of six are on error paths at driver initializations. This is not suprising

Table 3 Result. The table lists the overview of each bug we found. For precisely calculating the date of bug introductions (Since), we tracked the changes of file names.

Fixed file	Class	Faulty callback	Bug type	Path	Merged?	Since
drivers/power/88pm860x_charger.c	Generic	platform_driver::remove()	DoubleFree	Nromal	Yes	Jul 27 2012
drivers/media/pci/ddbridge/ddbridge-core.c	PCI	pci_driver::probe()	FreeRequestFailed	Error	Yes	Jul 3 2011
drivers/pcmcia/yenta_socket.c	PCI	pci_driver::probe()	Leak	Error	Yes	Apr 16 2005*
drivers/power/wm831x_power.c	Generic	platform_driver::probe()	FreeRequestFailed	Error	Yes	Aug 10 2009
drivers/usb/gadget/udc/fotg210-udc.c	Generic	platform_driver::probe()	FreeRequestFailed	Error	Yes	May 30 2013
drivers/clocksource/sh_mtu2.c	Generic	platform_driver::probe()	Leak	Error	Not yet	Apr 30 2009

*The beginning date of the Linux git repository. Thus, the lifetime of the bug in yenta_socket.c is longer than ten years.

drivers/power/88pm860x_charger.c

```
L739: pm860x_charger_remove(...)
L740: {
L741:   struct pm860x_charger_info *info = ...;
L742:   int i;
L743:
L744:   power_supply_unregister(info->usb);
-       free_irq(info->irq[0], info);
L746:   for (i = 0; i < info->irq_nums; i++)
L747:     free_irq(info->irq[i], info);
L748:   return 0;
L749: }
```

Fig. 10 Code snippet for a *DoubleFree* on a power charger.

because developers can check normal paths for their drivers by reloading on their sites. However, one driver calls free_irq() twice at the normal path for a driver removal. Three FreeRequestFailed shows the effectiveness of path-sensitive analysis to validate the balances of two APIs.

Suprisingly, all the bugs we found have existed since the driver was introduced into the Linux kernel. Thus, we detected bugs that have survived a large number of code reviews, testing, and production runs for three to ten years. The bugs we found potentially cause typical transient failures as discussed in Section 2.2.

Although our analyzer tries to prioritize emitted reports, there are a large number of false negatives. The 60 bug reports the analyzer detected contain one report for *DoubleFree* and three reports for *FreeRequestFailed* in Table 3 (=93.3% false positives). On the other hand, two *Leak* bugs in Table 3 appear in 177 *MayLeak* reports (= 97.5% false positives). The 294 *Escaped* reports do not contain bugs as far as we inspected. Thus, there are at most 525 false positives out of 531 reports (= 98.9% false positives). Obviously, the false positive rate is very high even when we focus on the 60 bug reports. However, the analyzer reduces manual inspections from 2287 drivers to 531 reports.

Figure 10 shows the double free on the driver for a power charger. The bug can be found by simple intra-procedural analysis, but detecting it also requires loop extractions. Our fix is to simply remove the redundant free_irq() before the loop.

Figure 11 shows a Leak bug in a Cardbus driver. Static analysis reported *MayLeak* on this bug because `socket->cb_irq` was potentially updated via a pointer `socket` in line 1256. At runtime, the failure can be manifested only when pcmcia_register_socket() fails and the device delivers an interrupt. Missing free_irq() lets the interrupt handler read from or write to resources freed by the error handling of pcmcia_register_socket(). Our fix is simply to add missing free_irq().

Interestingly, the example in Fig. 11 also has four other resource-release omissions in the error paths for the driver probe.

drivers/pcmcia/yenta_socket.c

```
L1143: static int yenta_probe(...)
L1144: {
L1145:   struct yenta_socket *socket;

L1233:   if (... || request_irq(socket->cb_irq, ...)) {
L1235:     socket->cb_irq = 0;
L1236:     setup_timer(...);
L1237:     mod_timer(&socket->poll_timer, jiffies + HZ);

L1243:   } else {
L1244:     socket->socket.features |= SS_CAP_CARDBUS;
L1245:   }

L1255:   dev_printk(KERN_INFO, &dev->dev,
L1256:       "..." ,cb_readl(socket, ...));

L1261:   ret = pcmcia_register_socket(...);
L1257:   if (ret == 0) {
L1259:     ret = device_create_file(...);
L1260:     if (ret == 0)
L1261:       goto out;
L1262:
L1263:       /* error path... */
L1264:       pcmcia_unregister_socket(&socket->socket);
L1265:   }
L1266:
+        if (socket->cb_irq)
+          free_irq(socket->cb_irq, socket);
L1267: unmap: /* pcmcia_register_socket failure */
L1268:   iounmap(socket->base);
L1275: out:
L1276:   return ret;
L1277:}
```

Fig. 11 Code snippet for a *Leak* bug on a Cardbus driver.

The maintainer found the problem and we fixed them as well as the IRQ leak. For example, a timer created by setup_timer(...) was not destroyed at the failure paths after the false condition of the first branch in Fig. 11. This implies that we can even detect other kinds of bugs that frequently co-occur at the same path of IRQ handling by validating IRQ handling.

Figure 12 shows a *FreeRequestFailed* bug. The code inappropriately unifies two error handling codes that are located after the goto label `fail1`. Our checker tracked every paths and detected free_irq() after request_irq() fails, while it is difficult to reproduce the problem with dynamic testing. After request_irq() in line 1600, the symbolic execution engine in the Clang Static Analyzer bifurcates (or forks) the execution of checking state transitions. The bifurcated analyzer executions independently check state transitions after the transition to *Requested* or *RequestFailed*. Our fix adds a branch condition before calling free_irq() because the driver should call free_irq() in the case where the driver in the *Requested* state encounters another failure in line 1610.

drivers/media/pci/ddbridge/ddbridge-core.c

```
L1563: wm831x_power_probe(...) {

L1600:   stat = request_irq(dev->pdev->irq, irq_handler,
L1601:     irq_flag, "DDBridge", (void *) dev);
L1602:   if (stat < 0)
L1603:     goto fail1;

L1610:   if (ddb_i2c_init(dev) < 0)
L1611:     goto fail1;

L1629: fail1:
L1630:   printk(KERN_ERR "fail1\n");
L1631:   if (dev->msi)
L1632:     pci_disable_msi(dev->pdev);
-        free_irq(dev->pdev->irq, dev);
+             if (stat == 0)
+                   free_irq(dev->pdev->irq, dev);
```

Fig. 12 Code snippet for a *FreeRequestFailed* bug.

6. Related Work

Since Engler et al. pioneered the idea of checking API usage rules in operating systems [1], there have been numerous work for extracting code patterns to be checked in particular code. Engler et al. propose searching for them in the form of pairs of functions that occur together frequently [9]. Yang et al. [13] extract system models from a source code by tracking system behavior. Lawall et al. [12] use the insights of experienced developers to extract Linux API protocols. Saha et al. [3] focus on resource release operations inside a function to reduce the number of false positives. However, existing work often focuses on generic bug patterns that appear on general software. In this work, we focus on the design and implementation of checking bugs that are derived from our prior work [15] to extract typical bugs from real bug reports in Linux.

Given the complete specification of coding rules, it is easier to recognize bug patterns because all deviations from the specification can be considered bugs. SeL4 [17] enables the formal verification of an entire code by designing a verification-friendly OS architecture. In Linux, device drivers can avoid bugs by automatic synthesis of a formal specification [18]. Jitk enforces formal proof techniques to ensure bug-free BSD packet filters in Linux [19]. Unfortunately, unstable in-kernel APIs in Linux prevent them from supporting a broad range of driver classes.

DDVerify [20], [21], [22] verifies 1642 properties of 31 device drivers in Linux 2.6.16 in over four hours [21]. On the other hand, our analyzer checks 2287 drivers in Linux 4.1 in seven hours. Their efforts result in detecting two real bugs of a spinlock and IO port misuses. Unlike DDVerify, our checkers are not based on any proofs of both Linux API usages and driver execution models. However, our result of five fixes shows that static analysis based on past mistakes can detect faults, although our checkers need to reduce a large number of false positives for practical uses.

In the case of Windows, the specification of the kernel interfaces to the device drivers is provided. Static Driver Verifier (SDV) [8] makes use of the specification to find bugs in Windows device drivers. The model represents invocations of all the classes of device drivers by Windows kernel so that SDV can invoke typical sequences of driver functions for each device event. SDVRP [23] generalizes the SDV concept with a more robust and performant analysis engine (SLAM2) [24], [25]. SDVRP allows developers to conduct symbolic model checking by specifying platform manager and API models for C modules. Our analyzer and driver lifecycle emulations run as a Clang plugin, but we can implement it with SDVRP by specifying each model (if the analyzer can handle GCC's C language extensions, which Linux code frequently uses). In particular, our idea of execution-flow emulation is similar to their module entry specifications. However, Linux driver interfaces inherently depend on its implementation, and thus, specialized code analysis like our automatic generation of injected code is necessary to scale to hundreds of drivers in the Linux kernel. In addition to the execution-flow emulation, the verification core is obviously more accurate and scalable than our checkers. However, the case of our experience to fix five real bugs implies that studying past patches like that in Section 2 leads to bug-finding tools focusing on error-prone patterns in operating systems.

7. Conclusion

We presented our experience for checking bug patterns that are derived from real reports. The checkers we developed succeeded in contributing five bug reductions in Linux device drivers. All the detected bugs were serious but hard to find due to their nondeterministic and rarely executed properties. We believe our case shows that utilizing software repositories is promising and should be further researched to enhance the future software quality.

However, there remains a gap between bug pattern recognitions and checker development to achieve practical fault avoidance. For example, all process for finding bugs such as identifying bugs and implementing static analysis are very labor-intensive. Also, even if they are automated in the future, the static checking spends too much time for checking a large number of bug patterns. Therefore, we believe this work is just the first step of utilizing repository mining to achieve advanced systems of static checking.

Acknowledgments We thank the anonymous reviewers for their helpful feedback. This work was partially supported by funding from JSPS Research Fellowships for Young Scientists.

References

[1] Engler, D., Chelf, B., Chou, A. and Hallem, S.: Checking System Rules Using System-specific, Programmer-written Compiler Extensions, *Proc. 4th Conference on Symposium on Operating System Design & Implementation* (*OSDI '00*) (2000).

[2] Renzelmann, M.J., Kadav, A. and Swift, M.M.: SymDrive: Testing Drivers Without Devices, *Proc. 10th USENIX Conference on Operating Systems Design and Implementation* (*OSDI'12*), pp.279–292 (2012).

[3] Saha, S., Lozi, J.-P., Thomas, G., Lawall, J.L. and Muller, G.: Hector: Detecting Resource-Release Omission Faults in Error-handling Code for Systems Software, *Proc. 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (*DSN '13*), pp.1–12 (2013).

[4] Chipounov, V., Kuznetsov, V. and Candea, G.: S2E: A Platform for In-vivo Multi-path Analysis of Software Systems, *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS XVI*), pp.265–278 (2011).

[5] Kadav, A., Renzelmann, M.J. and Swift, M.M.: Tolerating Hardware Device Failures in Software, *Proc. ACM 22nd Symposium on Operating Systems Principles* (*SOSP '09*), pp.59–72 (2009).

[6] Park, S., Lu, S. and Zhou, Y.: CTrigger: Exposing Atomicity Violation

Bugs from Their Hiding Places, *Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS XIV*), pp.25–36 (2009).
[7] Cadar, C., Dunbar, D. and Engler, D.: KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs, *Proc. 8th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '08*), pp.209–224 (2008).
[8] 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, *Proc. 1st ACM European Conference on Computer Systems 2006* (*EuroSys '06*), pp.73–85 (2006).
[9] Engler, D., Chen, D.Y., Hallem, S., Chou, A. and Chelf, B.: Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code, *Proc. 18th ACM Symposium on Operating Systems Principles* (*SOSP '01*), pp.57–72 (2001).
[10] Lu, L., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H. and Lu, S.: A Study of Linux File System Evolution, *Proc. 11th USENIX Conference on File and Storage Technologies* (*FAST'13*), pp.31–44 (2013).
[11] Wang, X., Zeldovich, N., Kaashoek, M.F. and Solar-Lezama, A.: Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior, *Proc. 24th ACM Symposium on Operating Systems Principles* (*SOSP '13*), pp.260–275 (2013).
[12] Lawall, J., Brunel, J., Palix, N., Hansen, R., Stuart, H. and Muller, G.: WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code, *Proc. IEEE/IFIP International Conference on Dependable Systems Networks* (*DSN '09*), pp.43–52 (2009).
[13] Yang, J., Twohey, P., Engler, D. and Musuvathi, M.: Using Model Checking to Find Serious File System Errors, *Proc. 6th Conference on Symposium on Opearting Systems Design & Implementation* (*OSDI '04*) (2004).
[14] Li, Z. and Zhou, Y.: PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code, *Proc. 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC/FSE-13*), pp.306–315 (2005).
[15] Yoshimura, T. and Kono, K.: Who Writes What Checkers?: Learning from Bug Repositories, *Proc. 10th USENIX Conference on Hot Topics in System Dependability* (*HotDep '14*) (2014).
[16] Pedro Fonseca, Rodrigo Rodrigues, B.B.B.: SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Exploration, *Proc. 11th USENIX Conference on Operating Systems Design and Implementation* (*OSDI'14*), pp.415–431 (2014).
[17] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an OS Kernel, *Proc. ACM 22nd Symposium on Operating Systems Principles* (*SOSP '09*), pp.207–220 (2009).
[18] Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E. and Heiser, G.: Automatic Device Driver Synthesis with Termite, *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles* (*SOSP '09*), pp.73–86 (2009).
[19] Wang, X., Lazar, D., Zeldovich, N., Chlipala, A. and Tatlock, Z.: Jitk: A Trustworthy In-kernel Interpreter Infrastructure, *Proc. 11th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '14*), pp.33–47 (2014).
[20] Witkowski, T., Blanc, N., Kroening, D. and Weissenbacher, G.: Model Checking Concurrent Linux Device Drivers, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE '07*), pp.501–504 (2007).
[21] Witkowski, T.: Formal Verification of Linux Device Drivers, Master's thesis, TU Dresden and ETH Zurich (2007).
[22] DDVerify: DDVerify, available from ⟨http://www.cprover.org/ddverify/⟩.
[23] Ball, T., Bounimova, E., Levin, V., Kumar, R. and Lichtenberg, J.: The Static Driver Verifier Research Platform, *Proc. 22nd International Conference on Computer Aided Verification* (*CAV '10*), pp.119–122 (2010).
[24] Ball, T., Bounimova, E., Kumar, R. and Levin, V.: SLAM2: Static Driver Verification with Under 4% False Alarms, *Proc. 2010 Conference on Formal Methods in Computer-Aided Design* (*FMCAD '10*), pp.35–42 (2010).
[25] SLAM: SLAM, available from ⟨http://research.microsoft.com/en-us/projects/slam/⟩.

Takeshi Yoshimura received his B.E. and M.E. degrees from the Department of Information and Computer Science at Keio University in 2011 and 2013, respectively. He is currently a Ph.D. student at the School of Science for Open and Environmental Systems at Keio University. His research interests include dependable systems, operating systems and virtualization. He is a member of ACM and USENIX.

Kenji Kono received his B.Sc. degree in 1993, M.Sc. degree in 1995, and PhD degree in 2000, all in computer science from the University of Tokyo. He is a professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.