Post-silicon Functional Validation with Virtual Prototypes

Kai Cong
Portland State University

Post-silicon Functional Validation with Virtual Prototypes

by

Kai Cong

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Portland State University
2015

ABSTRACT

Post-silicon validation has become a critical stage in the system-on-chip (SoC) development cycle, driven by increasing design complexity, higher level of integration and decreasing time-to-market. According to recent reports, post-silicon validation effort comprises more than 50% of the overall development effort of an 65nm SoC. Though post-silicon validation covers many aspects ranging from electronic properties of hardware to performance and power consumption of whole systems, a central task remains validating functional correctness of both hardware and its integration with software. There are several key challenges to achieving accelerated and low-cost post-silicon functional validation. First, there is only limited silicon observability and controllability; second, there is no good test coverage estimation over a silicon device; third, it is difficult to generate good post-silicon tests before a silicon device is available; fourth, there is no effective software robustness testing approaches to ensure the quality of hardware/software integration.

We propose a systematic approach to accelerating post-silicon functional validation with virtual prototypes. Post-silicon test coverage is estimated in the pre-silicon stage by evaluating the test cases on the virtual prototypes. Such analysis is first conducted on the initial test suite assembled by the user and subsequently on the expanded test suite which includes test cases that are automatically generated. Based on the coverage statistics of the initial test suite on the virtual prototypes, test cases are automatically generated to improve the test coverage. In the post-silicon stage, our approach supports coverage evaluation of test cases on silicon devices to ensure fidelity of early coverage evaluation. The generated test cases are issued to silicon devices to detect inconsistencies between virtual prototypes and silicon devices using conformance checking. We further extend the test case generation framework to generate and inject fault scenario with virtual prototypes for driver robustness testing. Besides virtual prototype-based fault injection, an automatic driver fault injection approach is developed to support runtime fault generation and injection for driver robustness testing. Since virtual prototype enables early driver development, our automatic driver fault injection approach can be applied to driver testing in both pre-silicon and post-silicon stages.

For preliminary evaluation, we have applied our coverage evaluation and test generation to several network adapters and their virtual prototypes. We have conducted coverage analysis for a suite of common tests on both the virtual prototypes and silicon devices. The results show that our approach can estimate the test coverage with high fidelity. Based on the coverage estimation, we have employed our automatic test generation approach to generate additional tests. When the generated test cases were issued to both virtual prototypes and silicon devices, we observed significant coverage improvement. And we detected 20 inconsistencies between virtual prototypes and silicon devices, each of which reveals a virtual prototype or silicon device defect. After we applied virtual prototype-based fault injection approach to virtual prototypes for three widely-used network adapters, we generated and injected thousands of fault scenarios and found 2 driver bugs. For automatic driver fault injection, we have applied our approach to 12 widely-used drivers. After testing all these drivers, we found 28 distinct bugs.

DEDICATION

To my parents, Yizi Cong and Xiuqin Bi

To my wife, Jin Zhang

ACKNOWLEDGMENTS

This dissertation could not have been accomplished without the generous help and support from many professors, colleagues, friends and my family. I would like to express my sincere gratitude to all of them.

First and foremost, I would like to express my thanks to my advisor Prof. Fei Xie for his insightful guidance, continuous support and endless encouragement. He is a great advisor. He has given me not only tremendous freedom to explore my industrial and research interests, but also opportunities to present at different conferences and work with people from different industry companies. He is a wonderful researcher. He has provided me consistently good advice and challenges which are great driving force to my Ph.D. research. His professional expertise with positive attitude has been always an inspiration to me.

I'm grateful to my committee members, Prof. Suresh Singh, Prof. Jingke Li, and Prof. Fu Li. They have provided many inspirational feedbacks for my research and valuable comments for my dissertation. I also want to express sincere appreciation for their time and effort in service on my doctoral committee.

Thanks to my fellow graduate students, Juncao Li, Kecheng Hao, Li Lei, Zhenkun Yang, Disha Puri, Dejun Qian, Christopher Havlicek, and Sharookh Daruwalla. Discussions with them have accelerated my Ph.D. study and broadened my understanding of related research domains.

Finally, I especially wish to acknowledge and thank my family for their love, support and constant encouragement. I am grateful to my parents, Yizi Cong and Xiuqin Bi, for all their love and guidance. Their diligent and optimistic characters have been always the role model for me. Last, but absolutely not least, I must thank my wife, Jin Zhang, for her unending love and encouragement. I can never thank her enough for her patience and understanding through the ups and downs of my Ph.D. study. I also would like to recognize the influence of my soon-to-be-born son, who has already brought me so much joy and hope.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 MOTIVATION AND PROBLEM STATEMENT

1.1.1 Motivation

New computer systems: smart phones, wearable devices, tablets, laptops, servers, etc. are entering the market place at an ever-accelerating pace. This brings enormous pressures on the product development teams to shorten the time-to-market. A recent study by International Business Strategies indicates that a 3-month delay to market reduces revenue by about 30% for chip manufacturers in general, and the penalty is even more severe for fast-evolving markets such as mobile devices [34]. To exacerbate the pressures, the complexities of these systems, both their hardware and software, are increasing significantly. Quoting a SoC architect for a mobile platform, "a state-of-the-art mobile platform is considered more complex than a server due to the many types of technologies it integrates while the product cycle is often as short as two years." A crucial stage in the product development cycle is post-silicon validation, i.e., validation conducted on actual devices or silicon prototypes with corresponding drivers. Post-silicon validation is a significant, fastest-growing component of validation cost. According to recent industry reports [59], post-silicon validation effort often consumes more than 50% of an 65nm SoC's overall design effort. This demands innovative approaches to speed-up post-silicon validation and reduce its cost.

Though post-silicon validation covers many aspects ranging from electronics properties of hardware to performance and power consumption of whole systems, a central task remains validating functional correctness of both hardware and its integration with software. Recently virtual prototypes are increasingly used in hardware/software development to enable driver development and validation at an early stage even before silicon prototypes become available [70]. An example is how Intel used virtual prototypes to enable driver development for their 40G Ethernet adapter (E40G) before the silicon prototype became available [62]. An E40G virtual prototype was created and used to test and validate the E40G driver being developed. Bugs were found in the driver using the E40G virtual device, even before the real E40G device became available. Since virtual prototypes are utilized as a transaction-level replacement for silicon devices to support driver development and validation, it is greatly desired to extend the effectiveness of virtual prototypes into post-silicon functional validation so that the major efforts invested can be fully utilized. We see major potentials of virtual prototypes in post-silicon functional validation of both hardware and its integration with software.

1.1.2 Problem statement.

This dissertation research is concerned with how to speed-up post-silicon functional validation with virtual prototypes for both hardware and software development. We observe four major challenges to achieving our goal:

- *Limited Silicon Observability and Traceability.* The silicon device is typically a black box. The amount of run-time information that can be retrieved from the device internal with build-in test circuitries and advanced logic analyzers is still quite limited. Such limited observability and traceability make post-silicon validation difficult.

- *Lack of Good Test Coverage Estimation.* There lacks good test coverage metrics over a silicon device. Therefore, it is difficult to assess the effectiveness of test cases and prioritize their application. In addition, coverage metrics rooted in hardware design are not well suited for testing the integration with software.

- *Lack of Early Test Readiness.* Test cases for post-silicon validation must be ready before a silicon device is available. The time-to-market after the device is first available can be as short as several weeks. Therefore, it is highly desired to avoid spending this precious time on preparing, debugging, and fixing test cases.

- *Lack of Effective Fault Injection for Driver Testing.* Device drivers are critical system components that operate or control devices. To ensure the system reliability, device drivers must tolerate all kinds of system situations, such as low resource situations, PCI bus errors and DMA failures. Therefore, it is necessary that different kinds of system scenarios and faults can be generated and injected to test the driver robustness.

1.2 PROPOSED SOLUTION

We propose an approach to accelerating post-silicon functional validation and reducing its cost with virtual prototypes. As shown in Figure 1.1, virtual prototypes play a central role in our approach which mainly support three components:

- *Coverage analysis:* While a silicon device is often a black box, its corresponding virtual prototype is a white box, i.e., its internal structures and workings

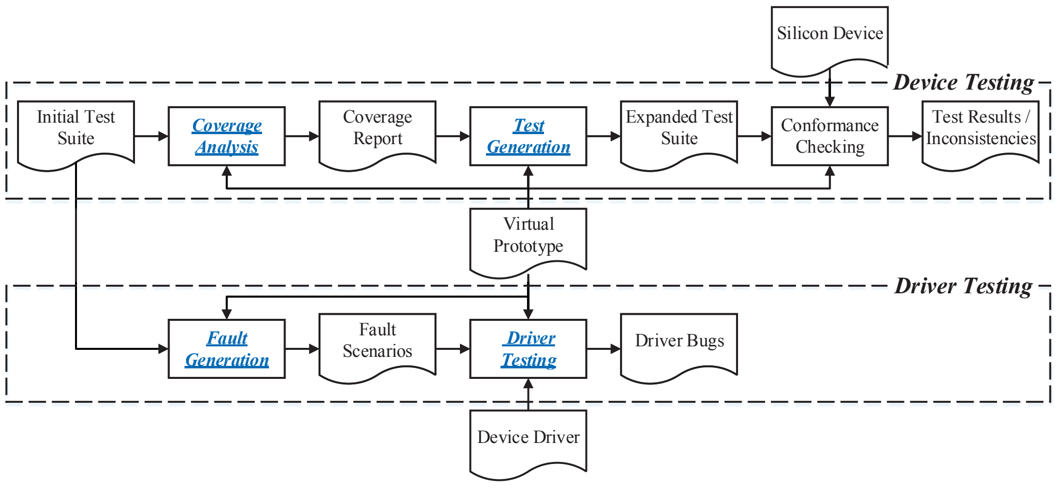


Figure 1.1: Main Components of Our Approach

are visible. The virtual prototype often models transaction-level behaviors of the silicon device. Therefore, the virtual prototype can be utilized to estimate the coverage of post-silicon validation tests on the functionalities of the silicon device.

- *Test generation:* Based on the coverage estimation, test cases can be automatically generated to specifically target silicon device functionalities that are yet covered. Test cases are particularly needed to trigger error handling conditions that are often hard to test with manually written tests. Expanded test suite can be used for testing silicon devices and supporting conformance checking.

- *Effective fault generation for driver testing:* Since virtual prototypes provide all device functionalities, many driver testing tasks can be conducted with virtual prototypes. We first develop a virtual prototype-based fault injection for driver robustness testing. Furthermore, we develop an automatic driver fault injection framework which can generate effective fault scenarios in a modest amount of time. The automatic driver fault injection framework can be applied to both virtual and silicon devices.

For coverage analysis and test generation, we employ symbolic execution of virtual prototypes as the foundation. More details about these components are elaborated below:

Symbolic Execution of Virtual Prototypes. The foundation of our approach is symbolic execution of virtual prototypes [20], utilized in calculating test coverage and generating new test cases. We have developed a symbolic execution environment (SEE) for QEMU virtual devices. Central to this environment is (1) how to encapsulate a virtual device in an execution harness that is sufficiently faithful to avoid crippling inaccuracy and sufficiently abstract to avoid prohibiting execution overheads and (2) how to reign in several limitations of symbolic execution by utilizing features of virtual devices.

Coverage Analysis of Post-silicon Tests. Test coverage is an important metric for evaluating the quality and readiness of post-silicon tests. We propose an online-capture offline-replay approach to coverage analysis of post-silicon validation tests with virtual prototypes for estimating silicon device test coverage [18]. We first capture necessary data from a concrete execution of the virtual prototype within a virtual platform under a given test, and then compute the test coverage by efficiently replaying this execution offline on the virtual prototype itself. Our approach provides early feedback on quality of post-silicon validation tests before silicon is ready. To ensure fidelity of early coverage evaluation, our approach have been further extended to support coverage evaluation and conformance checking in the post-silicon stage.

Automatic Concolic Test Generation. We present a concolic testing approach to generation of post-silicon tests with virtual prototypes [19]. This work is inspired by recent advances in concolic testing [30, 31]. Concolic (a portmanteau of concrete and symbolic) testing is a hybrid testing technique that integrates concrete execution with symbolic execution [39]. In our approach, we first identify device states under test from concrete executions of a virtual prototype using a transaction-based selection strategy, and then symbolically execute the virtual prototype from these states. Concrete tests are generated based on the symbolic path constraints obtained. We apply the generated test cases to both the silicon device and the virtual prototype, and check for inconsistencies between the real and virtual device states. Once an inconsistency is detected, we can replay the test case on the virtual prototype through symbolic execution to see whether it is a silicon device bug or a virtual prototype defect. The combination of virtual and silicon device execution brings three major benefits: (1) helping developers more easily and better understand a silicon device using its virtual prototype, (2) checking for defects in the silicon device, and (3) detecting bugs in the virtual prototype.

Effective Fault Injection for Driver Robustness Testing. Device drivers ought to be robust enough for handling different kinds of device faults instead of crashing or hanging the system. We first develop a virtual prototype-based fault injection approach by extending test case generation framework. The virtual prototype-based fault injection approach employs two fault models to generate device-related fault scenarios. After applying the approach to virtual prototypes of three widely-used network adapters, we have generated thousands of virtual prototype-based fault scenarios and triggered two driver crashes. Furthermore, to test the interfaces between device drivers and kernel API functions, we propose an automatic driver fault injection approach to generation and injection of fault scenarios with either virtual prototypes or silicon devices. Our approach runs a driver test and collects the corresponding runtime trace. Then we identify target functions which can fail from the captured trace, and generate effective fault s-cenarios on these target functions. Each generated fault scenario includes a fault configuration which is applied to guide further fault injection. Each fault scenario is applied to guide one instance of runtime fault injection and generate further fault scenarios. This process is repeated until all fault scenarios have been tested. To achieve systematic and effective fault injection, we have developed two key strategies. First, a bounded trace-based iterative generation strategy is developed for generating effective fault scenarios. Second, a permutation-based injection strategy is developed to assure the fidelity of runtime fault injection.

1.3 DISSERTATION OUTLINE

The remainder of this dissertation is organized as follows. Chapter 2 introduces a brief overview of background including virtual prototypes, symbolic execution and driver robustness testing. Chapter 3 presents symbolic execution of virtual prototypes which provides foundational support for our post-silicon functional validation. Chapter 4 elaborates coverage evaluation of post-silicon tests. Chapter 5 presents our approach to automatic concolic test generation. Chapter 6 illustrates automatic driver fault injection. Chapter 7 concludes and discusses future work.

Chapter 2

BACKGROUND

2.1 VIRTUAL PROTOTYPES AND QEMU VIRTUAL DEVICES

Virtual prototypes are fast, fully functional software models of hardware systems, which enable unmodified execution of software code. QEMU is a generic, open source machine emulator and virtualizer [10, 27]. We adopt QEMU virtual devices as the virtual prototypes for our study due to the open-source nature of QEMU and its wide varieties of virtual devices. Technology developed on QEMU virtual devices can be readily generalized to other open-source or commercial virtual prototyping environments due to their similarity in virtualization concepts, despite their different levels of modeling details.

To better understand the concept of virtual prototype, we illustrate it with a QEMU virtual device for the Intel E1000 Gigabit network adapter. The E1000 adapter is a PCI (Peripheral Component Interconnect) device which communicates with its control software through interface registers and interrupts. The E1000 virtual device has corresponding functions to support such communication, for instance, interface register functions and interrupt functions. In order to realize the functionalities of silicon devices, the E1000 virtual device also needs to maintain the device state and implement functions that virtualize device transactions and environment inputs. As shown in Figure 2.1, the E1000 virtual device has the following components:

```
// 1. Device state
typedef struct E1000State_st {
    PCIDevice dev; //PCI configuration
    uint32_t mac_reg[0x8000]; //Interface registers
    ......
    uint32_t rxbuf_size; //Internal variables
    ......
} E1000State;

// 2. Interface register function: write register
static void write_reg(void *opaque, uint64_t index, uint32_t value) {
    E1000State *s = (E1000State *)opaque;
    ......
    if(index == TRANSMIT) {
        s->mac_reg[index] = value;
        start_xmit(s); //Invoking transaction function
    }
    ......
}

// 3. Device transaction function: transmit packets
static void start_xmit(E1000State *s) {
    ......
    pci_dma_read(&s->dev, base, &desc, sizeof(desc)); //Invoking DMA functions
    ......
    set_irq(s->dev.irq[0],1); //Invoking interrupt function
}

// 4. Environment function: receive packets
static ssize_t receive(NetClientState *nc, const uint8_t *buf, size_t size) {
    ......
    pci_dma_write(&s->dev, base, &desc, sizeof(desc)); //Invoking DMA functions
    .....
    set_irq(s->dev.irq[0],1); //Invoking interrupt function
}
```

Figure 2.1: Excerpt of QEMU E1000 Virtual Device

- The device state, $E1000State$, which keeps track of the state of the E1000 device and the device configuration;
- The interface register functions such as *write_reg* which are invoked by QEMU to access interface registers and trigger transaction functions;
- The device transaction functions such as *start_xmit* which are invoked by the interface register functions to realize the functionality;
- The environment functions such as *receive* which are invoked by QEMU to pass environment inputs such as a packet received to the virtual device.

Both the device transaction functions and environment functions may access DMA data by calling DMA functions *pci_dma_write* and *pci_dma_read*, as well as fire interrupts by calling interrupt function *set_irq*. Both PCI interface functions and environment input functions are device entry functions which are invoked by QEMU to trigger device functionalities.

2.2 SYMBOLIC EXECUTION

Symbolic execution executes a program with symbolic values as inputs instead of concrete ones and represents the values of program variables as symbolic expressions. Consequently, the outputs computed by the program are expressed as a function of input symbolic values. The symbolic state of a program includes the symbolic values of program variables, a path condition, and a program counter. The path condition is a boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy for the symbolic execution to follow the particular path. The program counter points to the next statement to execute. A symbolic execution tree captures the paths explored by the symbolic execution of a program: the nodes represent the symbolic program states and the arcs represent the state transitions.

```
int f(int x)
{
  if(x < 0) return −x;
  if(x == 1) return 2;
  return x;
}
```

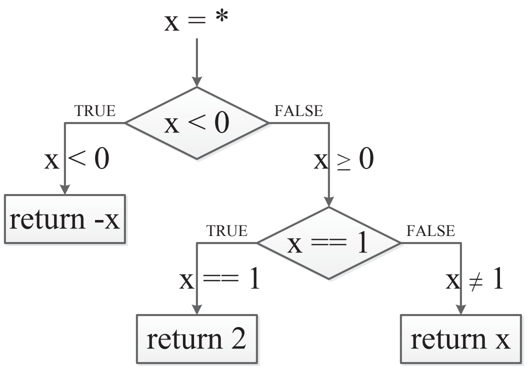


Figure 2.2: An Example of Symbolic Execution

We use the program in Figure 2.2 to illustrate how symbolic execution is conducted. At the entry, x has a symbolic value, i.e., any value allowed by its type (in this case, integer). At each branching point, the path condition is updated with conditions on the inputs to select between the two alternative paths. For this example, we can get three paths based on symbolic execution. Each path will have its own path condition, for example, $x < 0$ for the leftmost path.

2.3 POST-SILICON CONFORMANCE CHECKING

In previous work [43, 44, 45], we have developed an approach to post-silicon conformance checking of a silicon device with its virtual device. The conformance between the silicon and virtual devices is defined over their interface states. The request sequence issued to the device is first captured on the silicon device, and then replayed on the virtual device to check if the interface states of the silicon and virtual devices are consistent.

In Chapter 4 and 5, we utilize conformance checking for ensuring fidelity of coverage evaluation and evaluating efficiency of test generation results.

2.4 DRIVER ROBUSTNESS TESTING

According to the IEEE standard [1], robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions in software testing. The goal of robustness testing is to develop test cases and test environments where the robustness of a system can be assessed.

Kernel modules, especially device drivers, play a critical role in operating systems. It is important to assure that device drivers behave safely and reliably to avoid system crashes. Typically device drivers can work correctly under normal situations. However, it is easy for driver developers to mishandles certain corner cases, such as low resource situations, PCI bus errors and DMA failures.

```
int * p = (int *)kmalloc(size, GFP_ATOMIC);
p[10] = 3;
```

Figure 2.3: An Example with Kernel API Function Call

As shown in Figure 2.3, the *kmalloc* function is invoked to allocate a block of memory. After the function returns, the returned pointer is directly used without null pointer checking. Under normal system conditions, the *kmalloc* function returns successfully with a correct pointer to the allocated memory. However, when the *kmalloc* function returns a null pointer under a low resource situation, it is possible for the driver to crash the system. To handle such errors, the common approach is to add an error handling mechanism.

As shown in Figure 2.4, after the *kmalloc* function returns, the code checks whether the return value is a null pointer . If the *kmalloc* function returns a null pointer, the corresponding error handler is invoked to handle the error. However,

```
int * p = (int *)kmalloc(size, GFP_ATOMIC);
if(!p) goto error;
p[10] = 3;
......
error: error_handler();
```

Figure 2.4: An Example with Error Handler

a further concern is whether the error is handled correctly and does not trigger other driver or system errors.

To improve driver robustness, a device driver should be tested to see whether there exist two kinds of bugs: (1) driver error handling code does not exist; (2) driver error handling mechanisms do not handle the error correctly or trigger other driver/system issues. The first kind seems to be easy to avoid as long as driver developers write and check the code carefully. However, it still happens in the real world. The second kind is usually difficult and expensive to test.

2.5 RUNTIME DRIVER FAULT INJECTION

In driver robustness testing, all possible error conditions of a driver ought to be exercised. However, certain error conditions might be difficult and expensive to trigger, but efforts should be made to force or to simulate such errors to test the driver. Fault injection is a technique for software robustness testing by introducing faults to test code paths, in particular error handling code paths that, otherwise, might rarely be followed. Recently, fault injection techniques have been widely explored and studied for software testing and system robustness testing.

Runtime driver fault injection can be employed to simulate kernel interface failures to trigger and test error handling code. The common approach to driver

```
void * kmalloc(size_t size, int flags) {
    // Memory allocation operations
}


void * kmalloc_fault(size_t size, int flags) {
    return NULL;
}
```

Figure 2.5: A Driver Fault Injection Example

fault injection is to hijack the kernel function calls, such as *kmalloc* and *vmalloc*. By hijacking these functions, we can call the corresponding fault function to return a false result instead of invoking these functions. As shown in Figure 2.5, when *kmalloc* is invoked, the corresponding fault function $kmalloc\_fault$ is invoked to return a null pointer instead of a correct pointer to simulate an allocation error. In this way, we can test if device drivers can survive on different error handling code paths to improve driver robustness.

There are two main limitations with current driver fault injection. First, there is no automatic framework to support fault injection for different system function calls. Second, there is no a systematic test generation approach to generate effective fault scenarios. Currently most fault injections tools are using random fault injection which is facing major challenges in achieving desired effectiveness and avoiding duplicate fault scenarios.

In Chapter 6, we provide a framework which can automatically generate and inject fault scenarios at runtime. We have proposed a trace-based iterative generation strategy to produce unique and effective fault scenarios and developed a permutation-based replay mechanism to inject fault scenarios with high fidelity.

Chapter 3

SYMBOLIC EXECUTION OF VIRTUAL PROTOTYPES

3.1 OVERVIEW

Symbolic execution of virtual prototypes is the foundation for our approach to coverage evaluation and test generation. In order to symbolically execute virtual prototypes, we must address the following technical challenges:

- *Environment modeling.* A virtual device is not a stand-alone program. There are two issues with this incompleteness. First, the virtual device needs to be properly initialized and its entry functions properly exercised. Second, the virtual device may invoke libraries in its environment. Therefore, we need a solution to enclose the virtual device so that the symbolic execution engine can consume it and perform accurate and efficient analysis.

- *Symbolic execution engine adaptation.* We symbolically execute virtual devices using the KLEE symbolic execution engine. KLEE is not specially designed for executing virtual devices while virtual devices have specific characteristics. Hence, we need to adapt KLEE to execute virtual devices efficiently and provide more hardware-specific information.

Section 3.2 and 3.3 show the solutions for solving the above two challenges. Furthermore, we demonstrate that our approach employs symbolic execution engine to support both symbolic execution and analysis of virtual prototypes in Section 3.4.

3.2 HARNESS GENERATION

For symbolic execution of QEMU virtual devices, we adapt KLEE to handle the non-deterministic entry function calls and symbolic inputs to device models. Since the virtual device by itself is not a stand-alone program, in order for the symbolic engine to execute a virtual device, a harness must be provided for the virtual device. A key challenge here is how to create such a harness. This harness has to be faithful so that the symbolic execution of the virtual device will not generate too many paths infeasible in the real device. On the other hand, it has to be simple enough so that symbolic engine can handle the symbolic execution efficiently. To an extreme, the complete QEMU with the guest OS can serve as the harness which, however, is impractical for the symbolic engine to handle.

Currently we generate harnesses manually for major device categories. Since devices fall into device categories depending on interface types such as PCI and USB and on functionalities such as network adapters and massive storage devices, we started with creating harnesses for major device categories, e.g., PCI network adapters, and improved such a harness as we experiment on devices in this category. Manual harness generation involves examining how QEMU invokes the virtual device, what QEMU APIs that a virtual device invokes, and what these APIs invoke recursively, and deciding what to include. At times, it may be necessary to make an API produce non-deterministic outputs by throwing away its implementation. The harness includes the following parts as shown in Figure 3.1:

- Declarations of state variables and parameters of entry functions. A virtual device is not a stand-alone program. If a virtual device is running in a virtual machine, it will register its entry functions with the virtual machine. Moreover, the virtual machine will help the virtual device manage its state

```
//Declarations of necessary variables
E1000State state; //Device state
target_phys_addr_t address; //Address
......

int main() {
    //Load the concrete state
    load_state(&state, sizeof(state), "state");

    //Make parameters symbolic
    make_symbolic(&address, sizeof(address), "address");
    ......

    //Non-deterministic calls to entry functions
    switch(svd_deviceEntry) {
    case MMIO_WRITE:
        write_reg((void *)&state, address, value);
        break;
    case MMIO_READ:
        read_reg((void *)&state, address);
        break;
        ......
    }
}

//Stub functions
uint16_t net_checksum_finish(uint32_t sum) {
    ......
}
```

Figure 3.1: Excerpt of E1000 Virtual Device Harness

variables. Every time an entry function is invoked, the state variables and necessary parameters of the function will be made available to the function by the virtual machine. In order to exercise a virtual device symbolically, we need to handle the state variables and function parameters. Hence, we add declarations of state variables and inputs of entry functions to the harness.

- Code for loading the concrete state and making parameters of entry functions symbolic. In order to cover as many paths as possible in an entry function, we need to make certain inputs of the entry function symbolic. The inputs of an entry function contain state variables and necessary parameters. We implement two utility functions that are specially handled by the engine. Function "load_state" is used for loading the concrete state. Function "make_symbolic" is used for initializing the inputs symbolically.

- Non-deterministic calls to virtual device entry functions. For a real device, there are many ways for the OS and the environment to communicate with it. Similarly, virtual devices provide many types of entry functions for communicating with the OS and the environment. To analyze a virtual device, we go through all entry functions with symbolic inputs. We define a symbolic variable in the harness. With this symbolic variable, we make non-deterministic calls to all entry functions.

- Stub functions for virtual machine API functions invoked by virtual devices. Virtual devices often invoke API functions of virtual machines to achieve certain functionalities. Stubs for these functions need to be provided to complete the harness and are created manually as discussed above.

3.3 SYMBOLIC EXECUTION ENGINE ADAPTATION

To improve efficiency of symbolic execution, we modify KLEE to address four key technical challenges for symbolic execution of virtual devices.

3.3.1 Path Explosion Problem

Path explosion is a major limitation for symbolic execution to thoroughly test software programs. The number of paths through a program is roughly exponential in program size. The problem also exists in executing virtual devices symbolically.

We apply two constraints when executing the virtual device to combat the path explosion problem. First, we add a loop bound to each loop whose loop condition is a symbolic expression. With the loop bound, the user controls the depth of each loop explored. Currently, we add the loop bounds manually in virtual devices. This is practical since there are only a few loops in our analysis of three virtual devices. Second, we can add a time bound to ensure that symbolic execution will terminate in a given amount of time. If the symbolic execution does not complete within the given time bound, there may be unfinished paths. For such paths, we still generate test cases with path constraints obtained so far.

3.3.2 Environment Interaction Problem

A virtual device is a software component and may invoke outside API functions to interact with its environment. We divide such interactions into two categories based on whether a function call affects the values of variables in virtual devices. We detect whether the function has any pointer argument, accesses global variables, or returns a value. If so, this function potentially affects the values of variables in virtual devices. We then use two different mechanisms to handle functions in these two categories.

- If the function call does not affect the values of variables in virtual devices, we instruct KLEE to ignore it and issue a warning.
- If the function call may affect values of variables in virtual devices, we implement this function in our stubs. As there are not many such function calls for a category of virtual devices, such manual effort is acceptable.

3.3.3 Handling DMA

When a virtual device is processing a request, DMA data may be needed. QEMU provides two functions "pci_dma_read" and "pci_dma_write" for reading and writing DMA data separately. We ignore "pci_dma_write" function because it does not affect the device state. We instruct the symbolic execution engine to specially handle "pci_dma_read" function.

We hook "pci_dma_read" function to capture all run-time DMA read data in the concrete execution of the virtual device within the virtual machine. Then we utilize the captured data in both replay process and test generation process. In the replay process, every time "pci_dma_read" function is invoked, the corresponding data is loaded into the virtual device by the symbolic execution engine. In the test generation process, we compose a symbolic DMA sequence using the captured DMA data to guide test case generation.

3.3.4 Sparse Function Pointer Array Problem

A virtual device provides many different functions for realizing different device behaviors. For example, if a write register operation is issued to the virtual device, different functions can be triggered depending on different register offsets. Therefore, it is common for virtual devices to utilize a sparse function pointer array

```
//Declarations of a sparse function pointer array
static uint32_t (*macreg_readops[])(E1000State *, int) = {
    [RCTL] = mac_readreg, [TCTL] = mac_readreg, [ICS] = mac_readreg,
    [GPTC] = mac_read_clr4, [TPR] = mac_read_clr4, [TPT] = mac_read_clr4,
    [ICR] = mac_icr_read, [EECD] = get_eecd,  [EERD] = flash_eerd_read,
    ......
}
enum { NREADOPS = ARRAY_SIZE(macreg_readops) };

//Invoke the function using the function pointer
static uint64_t e1000_mmio_read(void *opaque, target_phys_addr_t addr,
    unsigned size)
{
    E1000State *s = opaque;
    unsigned int index = (addr & 0x1ffff) >> 2;

    if (index < NREADOPS && macreg_readops[index])
    {
        return macreg_readops[index](s, index);
    }
    ......
}
```

Figure 3.2: An Example of A Sparse Function Pointer Array

for accessing different functions, which makes the code concise. A sparse function pointer array is shown in Figure 3.2 which is used by QEMU E1000 virtual device.

If a symbolic execution engine invokes a function defined in the sparse function pointer array with a symbolic offset, the engine tries to explore all possible array offsets in order to cover all functions in the array. In this example, the symbolic engine needs to fork 5845 branches when the “macreg_readops” array is accessed. It takes much time to explore all 5845 branches. Actually only 7 functions are included in this function array. We summarize this information by static analysis of the virtual device. We modify the symbolic execution engine to specially handle sparse function pointer arrays. Every time a sparse function pointer array is accessed, we only fork branches according to the number of valid functions. In this example, we only fork 7 branches.

3.4 RUNTIME SHADOW EXECUTION

We first apply symbolic execution of virtual prototypes to runtime shadow execution to better understand runtime device state transitions. Runtime shadow execution allows the developer to monitor or diagnose a virtual device's behavior at runtime, ideal for the driver development and testing environment. Runtime shadow execution enables us to analyze run-time state transitions. We can follow the device sequence to trace all state transitions from the initial state. The detailed information of each state transition can be observed.

3.4.1 Runtime Shadow Execution Framework

To better understand device state transitions, we integrate symbolic execution of virtual devices into the virtual machine at runtime. The framework for runtime shadow execution is shown in Figure 3.3. The SEE interface has been implemented as the bridge between virtual machine and SEE. Our framework can support two mode: monitor and analysis modes.

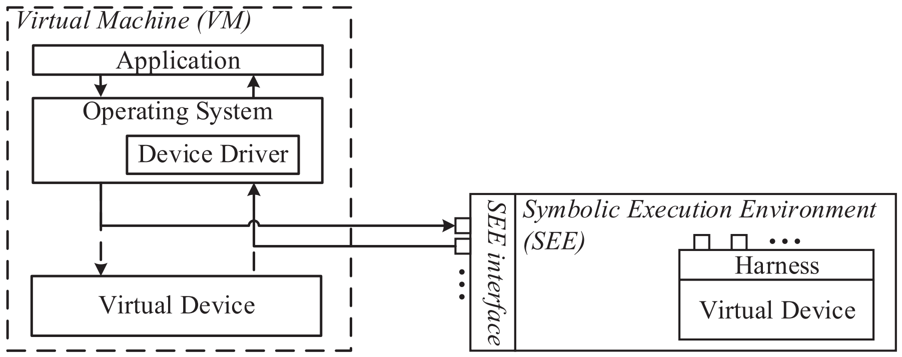


Figure 3.3: Framework for Runtime Analysis

Our runtime framework does not change the normal working process of the virtual machine. The framework only intercepts the communications between the virtual machine and the virtual device. Furthermore, our framework implements the SEE interface to act as the bridge transferring data from the virtual machine to SEE. The SEE interface intercepts three types of data:

1) Device states: It captures concrete device states when runtime shadow execution is enabled.

2) I/O requests and packets: It captures I/O requests and packets when there is a device request from either the driver or the environment.

3) DMA data: It captures DMA data when DMA data is accessed for processing a device request.

With the captured data, the virtual device can be executed concretely in monitor mode or symbolically in analysis mode.

3.4.2 Runtime Monitor Mode

In the monitor mode, the virtual device is executed concretely in both the virtual machine and SEE simultaneously. With the captured data through SEE interface, concrete execution can be conducted to enable runtime step-by-step analysis, which helps developers thoroughly understand the concrete state transition for processing a device request.

The harness for runtime monitor is slightly different from the harness for static analysis. An example of such harness is shown in Figure 3.4. Two special functions "*load_state*" and "*load_request*" are employed to load captured concrete device states and request information within the SEE. Then the corresponding entry function is invoked according to the request type.

Usually developers would like to analyze some desired state transitions. We provide two mechanisms to help developers select desired state transitions.

First, we provide a special user-level program to issue special I/O requests to label the start and finish points of a test case. The SEE interface parses all I/O requests. Once the special I/O requests are found, the SEE checks what kind of flag the request stands for. If it is a start flag, the SEE starts analyzing the upcoming requests. If it is a finish flag, the SEE stops analyzing the requests that follow.

3.4.3 Runtime Analysis Mode

The virtual device is executed concretely in the virtual machine and symbolically in the SEE simultaneously. The SEE executes the virtual device with the concrete device state and symbolic requests. It computes all the feasible execution paths under the current device state and generates runtime analysis test cases for the covered paths.

```
//Declarations of necessary variables
E1000State state; //Device state
target_phys_addr_t address; //Address
......

int main() {
    //Load the concrete device state
    load_state(&state, sizeof(E1000State), "state");

    //Load the concrete device request and request type
    load_request(&address, sizeof(address), "address");
    ......

    //Calls to interface functions
    switch(svd_deviceEntry) {
    case MMIO_WRITE:
        e1000_mmio_write((void *)&state, address, value);
        break;
    case MMIO_READ:
        e1000_mmio_read((void *)&state, address);
        break;
        ......
    }
}
```

Figure 3.4: Complete Harness for Runtime Monitor Mode

The harness for runtime analysis is the same as the one shown in Figure 3.1. Our approach assists developers in analyzing a virtual device symbolically at runtime. Once a request is selected or a breakpoint is hit in monitor mode, the virtual device can be executed symbolically with a symbolic request under the concrete state. All possible paths are explored. For each possible path, a runtime analysis test case is generated which contains the concrete device state and inputs that can be used to replay the corresponding path symbolically explored. Replaying the test case enables developers better observe and trace any variable change in the virtual device along this path. Furthermore, all paths explored at runtime are reachable. The developers can confirm what paths covered by static analysis can be covered at runtime. The developers can also alter the virtual device execution by injecting a device request identified by the SEE.

3.4.4 Further Potentials

This section illustrates how to employ symbolic execution of virtual prototypes to support runtime monitoring and analysis. There are two major functionalities provided by our symbolic execution approach.

First, it can thoroughly analyze each state transition and collect related information. It can support not only runtime monitoring but also coverage evaluation which is demonstrated in Chapter 4.

Second, it can execute a virtual prototype with a concrete state and symbolic requests. It can support not only runtime analysis but also test generation and fault generation which are demonstrated in Chapter 5.

Chapter 4

COVERAGE EVALUATION OF POST-SILICON VALIDATION TESTS

4.1 MOTIVATION AND OVERVIEW

Post-silicon validation has become a bottleneck in system development cycle and is a significant, growing part of overall validation cost [38]. To speed-up post-silicon validation, some tasks should be conducted early in the pre-silicon stage, e.g., development and evaluation of post-silicon validation tests. Test coverage is an important metric for evaluating the quality and readiness of post-silicon validation tests. Precise coverage results are necessary for engineers to judge whether existing test suites can achieve sufficient coverage and cover desired functionalities on the device.

Before the first silicon prototype is ready, it is very challenging to quantify coverage of post-silicon validation tests since we do not have a silicon device to run these tests on. Even if a silicon prototype is ready, the black box nature of the silicon prototype only supports limited observability and traceability that makes post-silicon validation difficult.

As shown in Figure 4.1, virtual prototypes and silicon devices are running respectively in virtual platforms and physical machines. Virtual prototypes can provide the same transaction-level functionalities as silicon devices to support driver development and validation. Virtual prototypes have major potential to play a crucial role in estimating silicon device functional coverage of post-silicon validation tests. The white box nature of virtual prototypes brings complete observability

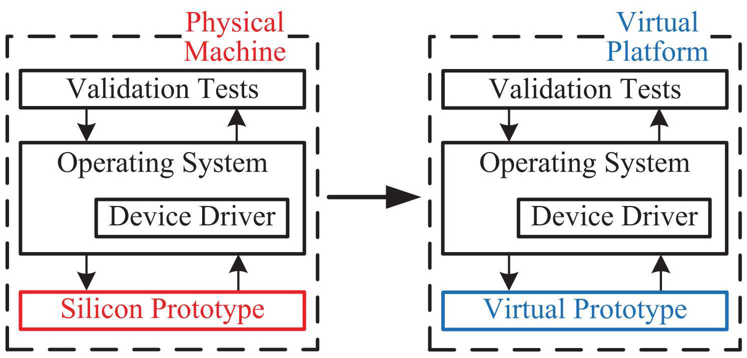


Figure 4.1: From Physical to Virtual

and traceability that evades silicon devices. It is possible to have thorough test coverage evaluation over virtual prototypes.

This chapter presents an online-capture offline-replay approach to coverage evaluation of post-silicon validation tests with virtual prototypes. We first capture necessary run-time data, including the initial device state and device requests from a concrete execution of the virtual prototype within a virtual platform under a given test. We then compute the test coverage by efficiently replaying captured data offline on the virtual prototype itself. To evaluate the coverage, we have adopted four typical software coverage metrics and developed two hardware-specific coverage metrics: register and transaction coverage. To ensure fidelity of coverage estimation on the silicon device, we further extend our approach to compute coverage after the silicon device becomes ready and check conformance with coverage estimate on the virtual prototype.

We have implemented this approach in Device Coverage Analyzer (DCA), a coverage analysis tool using virtual prototypes. We have applied our approach to evaluate a suite of common tests with virtual prototypes of five network adapters. Our approach was able to reliably estimate that this suite achieves high functional coverage on all five silicon devices.

4.2 PRELIMINARY DEFINITIONS FOR VIRTUAL DEVICES

In order to help better understand chapter 4 and 5, we introduce several definitions and define a formal model for a virtual device.

Definition 4.1. A ***device state*** is denoted as $\boldsymbol{s} = \langle \boldsymbol{s}_I, \boldsymbol{s}_N \rangle$ where $\boldsymbol{s}_I$ is the interface state including all interface registers and $\boldsymbol{s}_N$ is the internal state including all internal registers. The interface state $\boldsymbol{s}_I$ can be accessed by a high-level software (e.g., driver) while $\boldsymbol{s}_N$ is only accessed by the device itself.

As shown in Figure 2.1, the structure $E1000State$ represents the E1000 device state and includes interface registers mac_reg and an internal register $rxbuf\_size$.

Definition 4.2. An ***interface register request*** is denoted as $\boldsymbol{r}_{ir}$ which is issued by drivers to access interface registers.

Definition 4.3. An ***environment input*** is denoted as $\boldsymbol{r}_{ei}$ which is received by the device from the environment.

Definition 4.4. A ***device request*** is denoted as $\boldsymbol{r}$ which is issued by high-level software to control and operate the device.

As shown in Figure 2.1, the parameters *index* and *value* of interface register function $write\_reg$ can be treated as a request $\boldsymbol{r}$, which is issued by the driver to modify the interface register and trigger the transaction function.

Direct memory access (DMA) is a feature of modern computers that allows certain devices to access system memory independent of CPU. In order to process a device request $\boldsymbol{r}$, a device might read/write data using DMA.

Definition 4.5. A ***DMA sequence*** is denoted as $\boldsymbol{d} = \boldsymbol{d}_1, \boldsymbol{d}_2, ..., \boldsymbol{d}_n$ where $\boldsymbol{d}_i$ is the ith DMA data accessed for processing one request.

Definition 4.6. A ***device event*** is denoted as $\boldsymbol{e} = \langle \boldsymbol{r}, \boldsymbol{d} \rangle$ where $\boldsymbol{r}$ is a device request and $\boldsymbol{d}$ is a sequence of DMA data. For some event $\boldsymbol{e}$, $\boldsymbol{d}$ might be *null* since no DMA data is needed for processing $\boldsymbol{r}$.

Definition 4.7. A ***sequence of device events*** is denoted as $\boldsymbol{seq} = \boldsymbol{e}_1, \boldsymbol{e}_2, ...,$ $\boldsymbol{e}_n$. A subsequence $\boldsymbol{seq}_k$ of $\boldsymbol{seq}$ contains the first k events of $\boldsymbol{seq}$ where $\boldsymbol{seq}_k =$ $\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_k$. After processing a sequence of device events, the device can be transitioned to a new state from the initial state.

Definition 4.8. A ***test case*** is denoted as $\boldsymbol{tc} = \langle \boldsymbol{seq}, \boldsymbol{e} \rangle$, where $\boldsymbol{seq}$ is a sequence of device events and $\boldsymbol{e}$ is an additional device event. The device is transitioned to a desired state from the initial state after processing $\boldsymbol{seq}$. Then the device event e is issued to the device to trigger the desired device functionality.

Definition 4.9. A ***state under test*** is denoted as $\boldsymbol{s}_{ut}$ where $\boldsymbol{s}_{ut}$ is the device state on which test cases are generated.

Devices are transactional in nature: device requests are processed by device transactions. For a virtual device (which is a program), given a state s and a device request r, a program path of the virtual device is executed and the device is transitioned into a new state. Each distinct program path of the virtual device represents a distinct device transaction.

Definition 4.10. A ***device transaction***, denoted as $\boldsymbol{t} = \boldsymbol{l}_1, \boldsymbol{l}_2, ..., \boldsymbol{l}_n.$, is a program path of a virtual device. Each step $\boldsymbol{l}$ in the path is a tuple $(\boldsymbol{\lambda}, \boldsymbol{\gamma}, \boldsymbol{\xi})$, where $\boldsymbol{\lambda}$ is the code statement executed, $\boldsymbol{\gamma}$ is the registers accessed and $\boldsymbol{\xi}$ is the interrupt status.

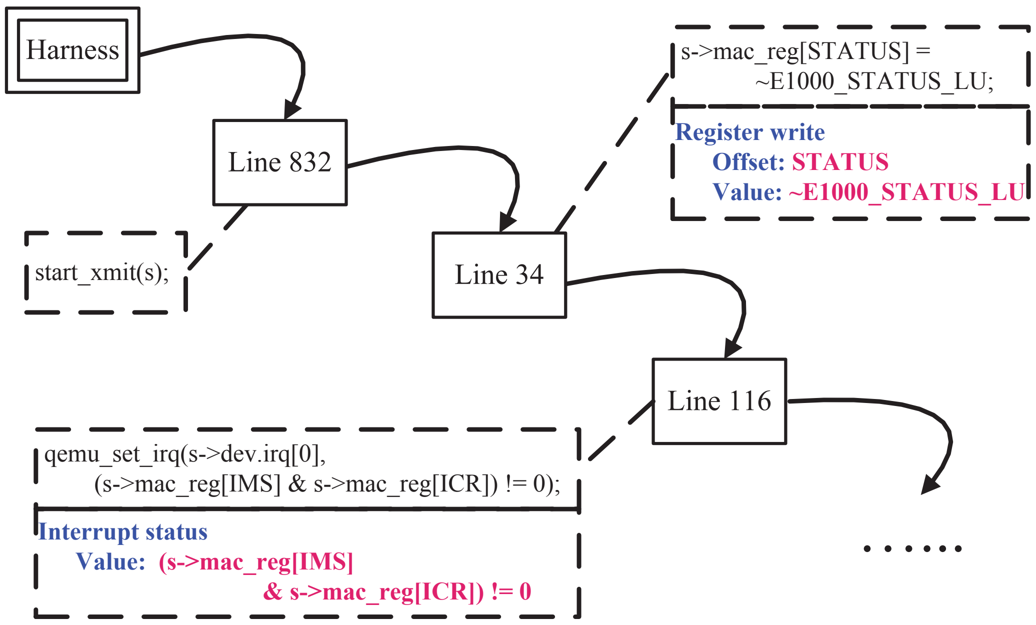

Harness
Line 832
start_xmit(s);
Line 34
s->mac_reg[STATUS] = ~E1000_STATUS_LU;
Register write
Offset: STATUS
Value: ~E1000_STATUS_LU
Line 116
qemu_set_irq(s->dev.irq[0], (s->mac_reg[IMS] & s->mac_reg[ICR]) != 0);
Interrupt status
Value: (s->mac_reg[IMS] & s->mac_reg[ICR]) != 0
......

Figure 4.2: A Transaction Example

Figure 4.2 gives an example of a transaction. Besides the basic code statement sequence, the transaction $\boldsymbol{t}$ also contains hardware-related information, such as the registers accessed and the interrupt status.

A virtual device is a transaction-level model of hardware design which can be represented as an event-driven state transition graph. As shown in Figure 4.3, given a device state s_{k-1} and a device event e_k, the device will transit to a new device state s_k. We use $s \xrightarrow{e} s'$ to denote a transaction.

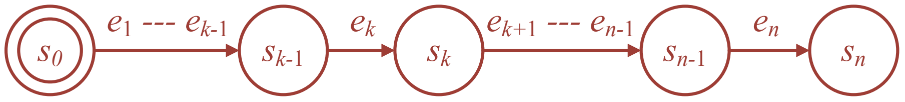

$$\begin{tikzcd} s_0 \arrow[r, "e_1 \text{ --- } e_{k-1}"] & s_{k-1} \arrow[r, "e_k"] & s_k \arrow[r, "e_{k+1} \text{ --- } e_{n-1}"] & s_{n-1} \arrow[r, "e_n"] & s_n \end{tikzcd}$$

Figure 4.3: A Graph Representation of State Transitions

4.3 ONLINE-CAPTURE OFFLINE-REPLAY COVERAGE EVALUATION

Before a silicon device is ready, post-silicon validation tests can be evaluated using RTL emulation. However, emulating hardware design has certain limitations. First, RTL emulators can be very expensive. Second, RTL emulation is often slow. Third, it requires a complete working RTL design [62] to evaluate post-silicon validation tests. Recently virtual devices and virtual platforms have been used for driver development and validation before a silicon device is ready. Virtual devices are software components. Compared to their hardware counterparts, it is easier to achieve observability and traceability on virtual devices. This makes virtual devices amenable to coverage evaluation of post-silicon validation tests.

4.3.1 Online-capture

In order to compute test coverage on virtual devices, we need to collect necessary run-time data from the virtual platform. A naïve idea is to capture all necessary run-time data including execution information of virtual devices directly from the virtual platform. However, such approach has three disadvantages. First, we need to instrument virtual devices to capture execution information of virtual devices. Second, capturing detailed execution information introduces heavy overhead into the virtual platform. Third, we need to decide what kinds of information should be captured before run-time execution of the virtual platform. It is hard to guarantee that captured information is sufficient. Once a new metric is added, it is possible that we have to modify the capture mechanism and then rerun the virtual platform to capture more data.

Therefore, we developed an online-capture offline-replay approach to capture minimum necessary data at run-time, and then replay the run-time data on the virtual device itself offline to collect necessary execution information.

A device can be treated as a state transition system. As shown in Figure 4.3, given a device state $\boldsymbol{s}_{k-1}$ and a device event $\boldsymbol{e}_k$, the device will transit to a new device state $\boldsymbol{s}_k$. Therefore, with the initial state $\boldsymbol{s}_0$ and the whole event sequence seq, we can infer all states and reproduce all state transitions. In other words, capturing $\boldsymbol{s}_0$ and seq from the concrete execution of a virtual device within the virtual platform should introduce the lowest overhead and deliver the most effective data.

4.3.2 Offline-replay

Our offline-replay mechanism reproduces run-time execution on virtual devices with $\boldsymbol{s}_0$ and seq, which provides flexible analysis mechanism and powerful debug capability.

Flexible analysis mechanism

The replay process is independent of the virtual platform/physical machine. Once run-time data is captured, users can replay the event sequence and reproduce the execution at any time. Based on different user requirements, users can generate different coverage reports from the replay process with different metrics.

Powerful debug capability

The replay mechanism provides capability for debugging interesting execution traces on virtual devices statement by statement, backward and forward.

Algorithm 1 REPLAY_EVENTS (s_0, seq)

```
1: $i \leftarrow 0$; //loop iteration
2: $s \leftarrow s_0$; //Set initial device state
3: while $i <$ seq.size() do
4:     $e \leftarrow seq[i]$;
5:     $\langle t, s_{next} \rangle \leftarrow$ Execute_Virtual_Device (s, e);
6:     T.save(t);
7:     $s \leftarrow s_{next}$; //Set next device state
8:     $i \leftarrow i + 1$;
9: end while
10: Generate_Report (T);
```

Algorithm 1 illustrates how to replay all events with $\boldsymbol{s}_0$ and seq to collect necessary execution information. In Algorithm 1, T is a temporary vector for saving execution information for all events. The algorithm takes the initial device state s_0 and the event sequence seq as inputs. Before replaying the event sequence, we set s_0 as the device state s. We run the virtual device with each event e in the event sequence seq and the corresponding state s to compute the execution information t and the next state s_{next}. Then t is saved in T and s_{next} is assigned to s. After replaying all events, we generate coverage reports based on T and user configuration.

4.3.3 Coverage Computation and Conformance Checking in the Post-silicon Stage

In our approach, we use coverage evaluation of virtual prototypes to estimate functional coverage on silicon devices. In order to make our approach practical and reliable, we need to address the following two key challenges:

1) Accuracy: In our approach, we capture run-time data from the concrete execution of virtual devices within a virtual platform. Events ($\boldsymbol{E}_v$) issued to virtual devices within a virtual platform can be different from events ($\boldsymbol{E}_s$) issued to silicon devices within a physical machine for the same tests. The concern is whether the coverage ($\boldsymbol{C}_v$) computed on ($\boldsymbol{E}_v$) is a good approximation of the coverage ($\boldsymbol{C}_s$) computed on ($\boldsymbol{E}_s$).

2) Conformance: Another challenge is whether coverage estimation on virtual devices can really reflect functional silicon device coverage. Although both virtual devices and silicon devices are developed according to the same specification, whether they conform to each other is still a major concern.

To address the above two challenges, we have extended our approach to support coverage computation and conformance checking after the silicon device is ready. We first reset the silicon device, and then capture run-time data, including all silicon device states $SS = \{ss_0, ss_1, ..., ss_n\}$ and the device event sequence $seq = e_1, e_2, ..., e_n$, from the concrete execution of a silicon device within a physical machine. For a silicon device, interface registers are observable while the internal registers are not observable in general. Therefore it is only possible to record all silicon device interface states $SS_I = \{ss_{I0}, ss_{I1}, ..., ss_{In}\}$ due to the limited observability. Algorithm 2 shows the extended algorithm for replaying SS_I and seq on the virtual device.

In Algorithm 2, we first reset the virtual device to get the initial device state s.

Algorithm 2 EXTENDED_REPLAY_EVENTS (SS_I, seq)

```
1: $k \leftarrow 0$; //loop iteration
2: $s \leftarrow Reset\_Virtual\_Device\ ()$; //$s = \langle s_I, s_N \rangle$
3: while $k < seq.size()$ do
4:     $s_I \leftarrow ss_{Ik}$; //Load captured silicon device interface state
5:     $e \leftarrow seq[k + 1]$;
6:     $\langle t,\ s' \rangle \leftarrow Execute\_Virtual\_Device\ (s,\ e)$; //$s' = \langle s'_I, s'_N \rangle$
7:     $T.save(t)$;
8:     $Check\_Conformance\ (s'_I,\ ss_{I(k+1)})$;
9:     $s_N \leftarrow s'_N$;
10:    $k \leftarrow k + 1$;
11: end while
12: $Generate\_Report\ (T)$;
```

We assume that the internal states between the silicon device and its virtual device are the same after resetting devices. Even if both internal states are not exactly the same, a few differences should not cause a large number of functional differences according to device specifications. We take the captured device state ss_{Ik} and e_{k+1} as inputs to replay one event. The virtual device is executed with s and e_{k+1} to compute the execution information and the state s' after processing e_{k+1}. Then conformance checking is conducted between the computed interface state s'_I on the virtual device and the captured interface state $ss_{I(k+1)}$ on the silicon device to detect inconsistencies. After replaying one event, we keep the internal state and load next interface state captured to compose the device state. After replaying all events, we can get coverage reports and inconsistency report.

We utilize the coverage evaluation and conformance checking results in three aspects to assure the coverage estimation accuracy. First, we compare $\boldsymbol{C}_s$ and $\boldsymbol{C}_v$

to detect differences. If we can verify that there is no difference or few differences between $\boldsymbol{C}_v$ and $\boldsymbol{C}_s$, we can better trust that $\boldsymbol{C}_v$ can be a good approximation of $\boldsymbol{C}_s$. Second, the number of inconsistencies provides basic measurement how many differences there are between the silicon device and the virtual prototype. After analyzing the inconsistencies, we further evaluate whether these inconsistencies cause different device behaviors. If there are few inconsistencies found and there is no significant effect on the device, it can increase our confidence on coverage estimation. Third, it is easy to fix the detected inconsistencies on the virtual device so that the fixed virtual device conforms with the silicon device. Then we compute coverage again on the fixed virtual device using the same test cases. By comparing the coverage report on the fixed virtual device with that on the silicon device, we further verify that the differences in coverage caused by the inconsistencies are removed.

4.4 COVERAGE METRICS

Computing test coverage requires appropriate coverage metrics. In our approach, we use virtual prototype coverage to estimate silicon device functional coverage. A virtual prototype is not only a software program, but also models the characteristics of the silicon device. Therefore we have employed two kinds of coverage metrics: we have adopted the typical software coverage metrics and developed two hardware-specific coverage metrics: register coverage and transaction coverage.

4.4.1 Code Coverage

Code coverage is a typical measure used in software testing. Virtual devices are software models. We can apply all code coverage metrics to virtual devices. We select four common coverage metrics: function coverage, statement coverage, block coverage and branch coverage.

4.4.2 Register Coverage

A hardware register stores bits of information in such a way that systems can write to or read out from it all the bits simultaneously. High-level software can determine the state of the device by reading registers, and control and operate the device by writing registers. It is critical for engineers to know what registers have been accessed so they can check whether the device is accessed correctly according to the specification. Virtual devices provide complete observability, therefore we can capture accesses on both interface and internal registers. Actually in our approach, we capture all register accesses and deliver different kinds of register coverage reports according to user configuration.

4.4.3 Transaction Coverage

Devices and, therefore, virtual devices are transactional in nature: they receive interface register requests and environment inputs, and process them concurrently without interference. Thus, an interesting and useful metric is transaction coverage. For a virtual device (which is a C program), given a state s and a device request r, a program path of the virtual device is executed and the device is transitioned into a new state. Each distinct program path of the virtual device represents a distinct device transaction. When computing coverage, the impact of a test case on the virtual device in term of what transactions it hits and how often they are hit are recorded. The impact of a test suite can be recorded the same way. The coverage statistics can be visualized using pie or bar charts in term of what and how many requests were made, what and how many transactions were hit, and what percentages they account for among all requests. Moreover, the details of a transaction is recorded, such as registers accessed and interrupt status.

4.5 IMPLEMENTATION

As shown in Figure 4.4, we first capture necessary data from a concrete execution of the virtual prototype within a virtual platform under a given test, and then compute the test coverage by efficiently replaying this execution offline on the virtual prototype itself. Our approach provides early feedback on quality of post-silicon validation tests before silicon is ready.

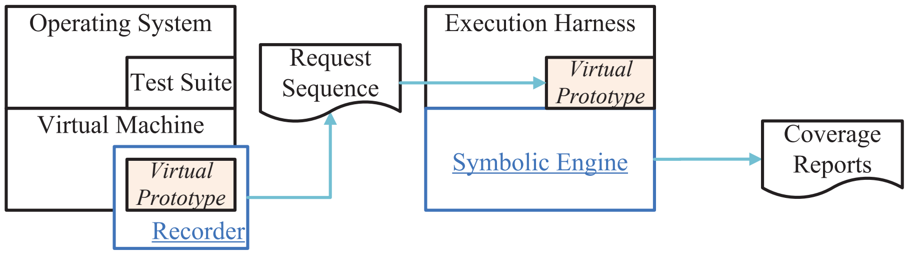


Figure 4.4: Workflow for Coverage Evaluation

4.5.1 Coverage on Different Levels

To generate coverage reports, we first analyze virtual devices statically to get program information, such as the position of branches and the number of functions, and then generate all kinds of coverage reports based on the execution traces computed by the replay engine. Our approach provides flexibility to generate reports on two different levels:

1) Event Level: Given an event, a user can check what transaction is explored, what registers are accessed and whether any interrupt is fired. Moreover, the user can debug the execution trace step by step using the replay engine.

2) Test Case/Suite Level: A test case/suite issues a sequence of requests to a device. Simultaneously, the device may receive environment inputs and read DMA data. Given a test case/suite, all device events are captured. The replay engine replays all captured events and generates the code coverage, the register coverage and the transaction coverage for the test case/suite.

4.5.2 Implementation Details

We implement our approach on the QEMU virtual platform. The event capture mechanism is implemented as a QEMU module which can be used for hooking QEMU virtual devices. Device interface functions are invoked by the QEMU framework. For instance, a driver issues a read register request, the QEMU invokes the corresponding read register function defined in the virtual device. Our module hooks all the interface functions when the virtual device registers these functions to QEMU. In this way, the module captures the device events when there is an interface register request, an environment input or a DMA access. This module provides capability to hook different virtual devices without modifying virtual devices. For capturing events on silicon devices in physical machines, we modified device drivers to achieve it.

We construct our replay engine using the symbolic execution engine KLEE [17]. We modify KLEE in three aspects. First, we implement some special function handler for loading events and DMA data. Second, we capture execution trace during execution of virtual devices. Third, we realize our own module for coverage generation.

4.6 EXPERIMENTAL RESULTS

We have applied DCA to QEMU-based virtual devices for five popular network adapters: Intel E1000, Broadcom Tigon3, Intel EEPro100, AMD RTL8139 and

Realtek PCNet. While our tool currently focuses on QEMU-based virtual devices, the principles also apply to other virtual prototypes. The experiments were performed on a desktop with an 8-core Intel(R) Xeon(R) X3470 CPU, 8 GB of RAM, 250GB and 7200RPM IDE disk drive and running the Ubuntu Linux OS with 64-bit kernel version 3.0.61.

4.6.1 Online-capture and Offline-replay Overhead

In order to evaluate our approach, we capture a request sequence triggered by a test suite. The test suite includes most common network testing programs, such as ifconfig and ethtool [19]. DCA needs to capture the initial device state and device events at run-time, which brings overhead to run-time QEMU environment. With the capture mechanism, both QEMU and virtual devices work normally.

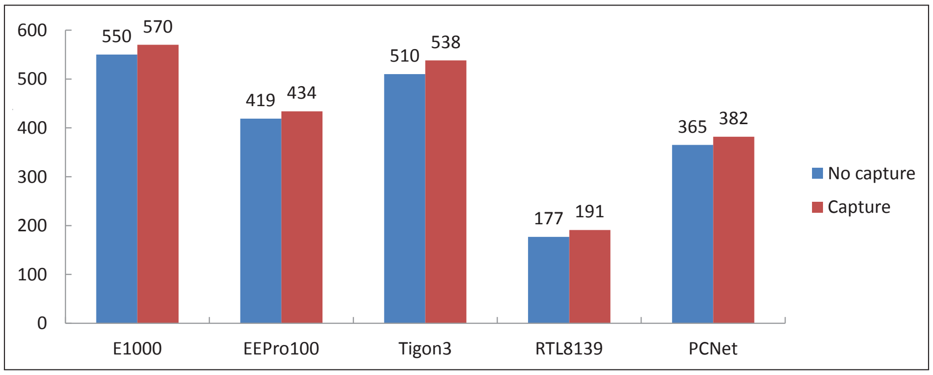


Figure 4.5: Time Usage (Seconds) for Online Capture

To evaluate the overhead of online capture mechanism, we illustrate the time usage for the whole test suite under the capture configuration and no-capture configuration in Figure 4.5. Between the capture and no-capture configurations, there is low running time overhead introduced. For example, the overhead for E1000 is about (570 - 550) / 550 = 3.6%.

We further evaluated time and memory usages for the offline replay process.

As shown in Table 4.1, time and memory usages of the offline replay are modest. It only takes a few minutes to process tens of thousands events.

Table 4.1: Time and Memory Usages for Offline Replay

	Events(#)	Time(Minutes)	Memory(Mb)
E1000	65530	10.5	268.24
Tigon3	89032	12.0	336.35
EEPro100	30112	6.0	213.18
RTL8139	43228	7.0	225.26
PCNet	54016	8.5	254.60

4.6.2 Coverage Results

We demonstrate our coverage results in three aspects: code coverage (statement/block/branch/function coverage), register coverage and transaction coverage. Due to space limitation, we only illustrate coverage results for E1000 below although we have finished coverage evaluation on all five devices.

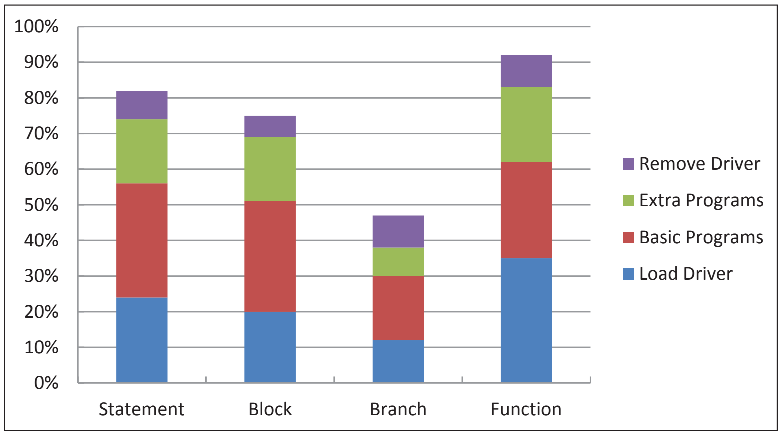


Figure 4.6: Code Coverage Results for E1000

Figure 4.6 uses a stack to show incremental coverage of different test programs on E1000 under different code coverage metrics. We evaluate the coverage for both a test case, such as sending a ping packet, and a test suite including most common testing programs. These coverage results can give engineers basic measurement of the quality of test cases.

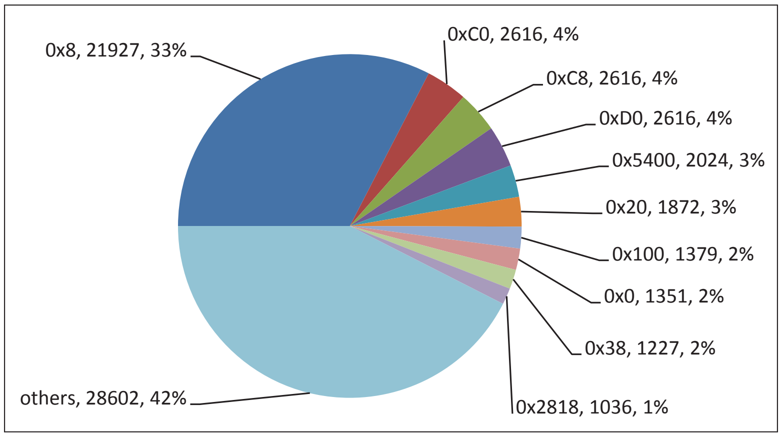


Figure 4.7: Top Ten Accessed Registers for E1000

Figure 4.7 shows partial register coverage results for E1000. Each register is identified using the register offset, such as 0x0 and 0x8. The figure shows that how many times and how much percentage top ten registers are accessed. For instance, the most accessed register is register 0x8 (status register), which is accessed 21927 times. The system software reads this register very frequently to query the device state.

Figure 4.8 shows partial transaction coverage results for E1000. Each transaction is identified using a hash value, such as 0xd4e4d3ed. It shows that how many times and how much percentage top ten transactions are accessed. By analyzing transaction coverage, engineers can know what functionalities have been tested. By analyzing execution information of each transaction, engineers can further observe register accesses.

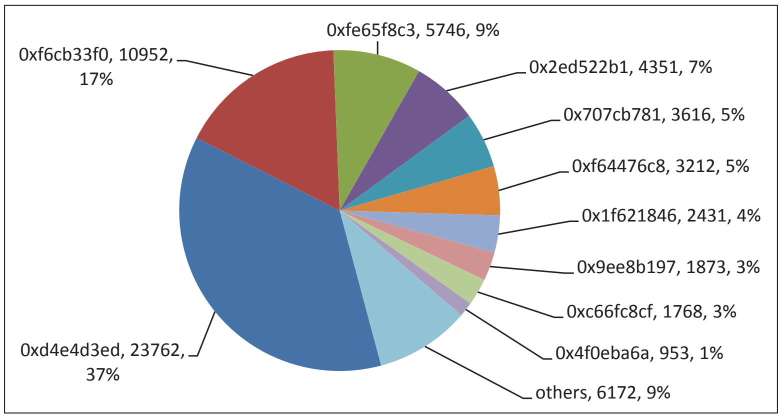


Figure 4.8: Top Ten Transactions for E1000

4.6.3 Coverage and Conformance Results in Post-silicon Stage

With the same test suite, we instrumented drivers to capture run-time data on two silicon devices: E1000 and Tigon3, and computed the coverage on the corresponding virtual devices. We compare the results with these results shown in Section 4.6.2. The coverage results are very similar for both E1000 and Tigon3 in terms of code and register coverage. One major difference is reflected on transaction coverage. Due to different speeds of physical machine and virtual platform, several transactions are affected. For example, while transmitting network packets, silicon devices can transmit more packets than virtual devices in the transmit transaction since the speed of silicon devices is much higher than virtual devices. We conclude such differences in coverage are acceptable.

We applied conformance checking to detect inconsistencies between E1000 and Tigon3 and their corresponding virtual devices. There are 13 inconsistencies discovered between the two network adapters and their virtual devices under the given tests: 7 in Intel E1000 and 6 in Broadcom BCM5751. We modified 21 lines of code in virtual devices to fix all 13 inconsistencies. Then we rerun coverage tools on fixed virtual devices to generate new coverage reports. After comparing the new reports with the post-silicon coverage reports, we found no differences except the known transaction differences.

Remarks: Coverage evaluation in the post-silicon stage often requires instrumenting the device driver and comes too late. Coverage evaluation on virtual prototypes can be available much earlier; therefore, it can guide improvement of post-silicon tests. From conformance checking results and coverage report comparison, it is clear the more conforming the virtual and silicon devices are, the more accurate the coverage evaluation on the virtual device. Even if there exist inconsistencies, conforming checking facilitates quick correction of coverage estimate in the post-silicon stage by conveniently detecting these inconsistencies.

4.7 RELATED WORK

One common approach to post-silicon coverage evaluation is to use in-silicon coverage monitors [8, 12, 50]. However, adding coverage monitors to the silicon is costly in terms of timing, power, and area [3]. In order not to introduce too much overhead, developers can only add a small number of coverage monitors in the design. Consequently, the effectiveness of coverage evaluation highly relies on what kinds of device signals are captured by in-line coverage monitors. Moreover, such approach of using coverage monitors can take effect only after silicon devices are ready. Another approach to coverage evaluation of test cases before silicon devices are available is RTL emulation. However, emulating hardware design has some limitations as we discussed in 5.2.1. Our approach takes the obvious advantages of virtual devices: complete observability and traceability, and is applicable without silicon devices. We utilize test coverage over virtual devices to estimate silicon device functional coverage.

4.8 SUMMARY

Quantifying coverage of post-silicon validation tests is very challenging due to limited hardware observability [57]. In this chapter, We have presented an approach to early coverage evaluation of post-silicon validation tests with virtual prototypes, which fully leverages the observability and traceability of virtual prototypes. We have applied our approach to evaluate a suite of common tests on virtual prototypes of five network adapters. We have also established high confidence in fidelity of coverage evaluation by further conducting coverage evaluation and conformance checking on silicon devices.

Chapter 5

AUTOMATIC CONCOLIC TEST GENERATION

5.1 MOTIVATION AND OVERVIEW

To accelerate post-silicon validation, high-quality tests should be ready before a silicon device is available [57]. The time-to-market after the device is first available can be as short as several weeks. Therefore, it is highly desired to avoid spending this precious time on preparing, debugging, and fixing tests. There should be high-quality tests available before the first silicon prototype is ready.

Currently, tests for post-silicon functional validation mainly include random tests, manually written tests, and end-user applications [33][79]. Random testing can quickly generate many tests and is easy to use while facing major challenges in achieving high coverage of device functionalities and avoiding high redundancy in tests. Manually written tests are efficient in testing specific device functionalities. However, developing manual tests is labor-intensive and time-consuming. Furthermore, humans make mistakes when they write tests manually and it is difficult to check correctness of these tests until they are applied to a silicon device. End-user applications are convenient and easy to deploy; however, it is often difficult to quantify what device functionalities are covered. In addition, end-user applications are generally not device-specific, therefore often leading to insufficient coverage of device functionalities.

Recently virtual prototypes are increasingly used in hardware/software co-development to enable early driver development and validation before hardware devices become available [62, 70]. Virtual prototypes also have major potential to play a crucial role in test generation for post-silicon validation.

This chapter presents a concolic testing approach to automatic post-silicon test generation with virtual prototypes. This work is inspired by recent advances in concolic testing [30, 31]. Concolic (a portmanteau of concrete and symbolic) testing is a hybrid software testing technique that integrates concrete execution with symbolic execution [39]. In our approach, we borrow "concolic" literally and conduct concolic test generation with virtual prototypes by integrating concrete runtime execution and symbolic execution. We first identify device states under test from concrete executions of a virtual prototype using a transaction-based selection strategy, and then symbolically execute the virtual prototype from these states. Concrete tests are generated based on the symbolic path constraints obtained. We apply the generated test cases to both the silicon device and the virtual prototype, and check for inconsistencies between the real and virtual device states. Once an inconsistency is detected, we can replay the test case on the virtual prototype through symbolic execution to see whether it is a silicon device bug or a virtual prototype defect. The combination of virtual and silicon device execution brings three major benefits: (1) help developers more easily and better understand a silicon device using its virtual prototype, (2) check for defects in the silicon device, and (3) detect bugs in the virtual prototype.

We have implemented our approach in a prototype post-silicon test generation tool, namely, ACTG (Automatic Concolic Test Generation). We have applied ACTG to virtual prototypes for three widely-used network adapters. ACTG generates hundreds of unique tests for each device. These tests lead to significant improvement in coverage. When applying the generated test cases to the silicon devices, ACTG detects 20 inconsistencies between the virtual and silicon devices.

Our approach makes the following key contributions:

- *Concolic testing for post-silicon validation.* Our approach to post-silicon device test generation not only integrates concrete and symbolic execution, but also combines virtual and silicon device executions. The observability and controllability of virtual prototypes are fully leveraged while generated tests are compatible with silicon devices.

- *Transaction-based test selection.* A transaction-based test selection strategy is developed to select device states under test and eliminate redundancy in generated tests. This strategy not only helps generate test cases with high functionality coverage in modest amount of time, but also produce efficient test cases with low redundancy.

5.2 CONCOLIC TEST GENERATION WITH VIRTUAL PROTOTYPES

5.2.1 A Naïve Approach

Virtual devices are software components. Compared to their hardware counterparts, it is easier to achieve observability and traceability on virtual devices. This makes virtual devices amenable to post-silicon test generation.

A naïve approach to test generation with a virtual device is to apply symbolic execution directly to it. A virtual device can be treated as an event-driven program as shown in Figure 4.3. A virtual device processes a possibly unbounded sequence of events from the initial state. In other words, a virtual device can be abstracted as a program that has an infinite loop as shown in Figure 5.1.

Within the main function, the device state is first set to the initial state by resetting the device. Then, there is an infinite loop which is used for handling

```
E1000State state; //Device state

int main() {
    // Reset the device
    device_reset(&state);

    while(1) {
        /* Read the next incoming event. Usually this is treated as a blocking
            function. */
        EVENT event = read_next_device_event();

        /* Handle the event based on the current state. */
        /* The corresponding entry function is invoked, e.g. write_reg(...) ->
            start_xmit(...) */
        switch(event.type) {
        case MMIO_WRITE:
            write_reg((void *)&state, event.address, event.value);
            break;
        case MMIO_READ:
            read_reg((void *)&state, event.address);
            break;
            ......
        }
    }
}
```

Figure 5.1: Abstract Event-driven Model of QEMU E1000 Virtual Device

device events. The loop body can be clearly divided into two stages. First, it reads the next device event if there is one. Second, it invokes the correct entry function to process the event under the current state. After the event is processed, the device state is still saved in s and continue to process the next device event.

To execute such a virtual device symbolically, we first set the device reset state as the initial state s_0 of the virtual device, which is a concrete state. Then we symbolically execute the virtual device from s_0 with a sequence of symbolic device events.

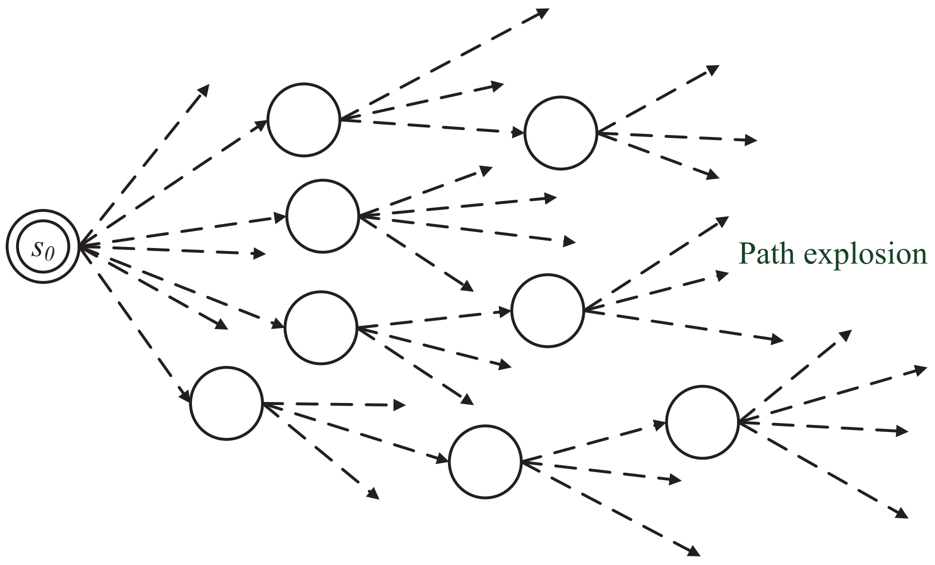


Figure 5.2: Path Explosion Problem

Such execution can easily lead to a path explosion [13, 41] as shown in Figure 5.2 due to the following two reasons. First, the abstract model of a virtual device shown in Figure 5.1 includes an infinite loop. Second, for each loop iteration, it introduces a new symbolic event, which means each iteration produces many new paths. After processing a sequence of symbolic events, the number of paths increase exponentially. Indeed as we tried this approach, it caused a path explosion only after processing a few symbolic device requests. Moreover, most functionalities of the virtual device are only triggered by long, well-formed sequences of requests from the reset state. Therefore, the above naïve approach cannot generate deep test sequences that sufficiently cover device functionalities.

5.2.2 Concolic Test Generation Algorithm

In order to address the challenge in Section 5.2.1, we develop a concolic testing scheme that integrates both concrete and symbolic execution. Concrete execution is first carried out on the virtual device and a sequence *seq* of concrete events issued to the device by the driver is captured. With *seq*, a set of device states can be computed on the virtual device, as shown in Figure 5.3 where solid arrows denote concrete device execution while dashed arrows denote generated test cases.

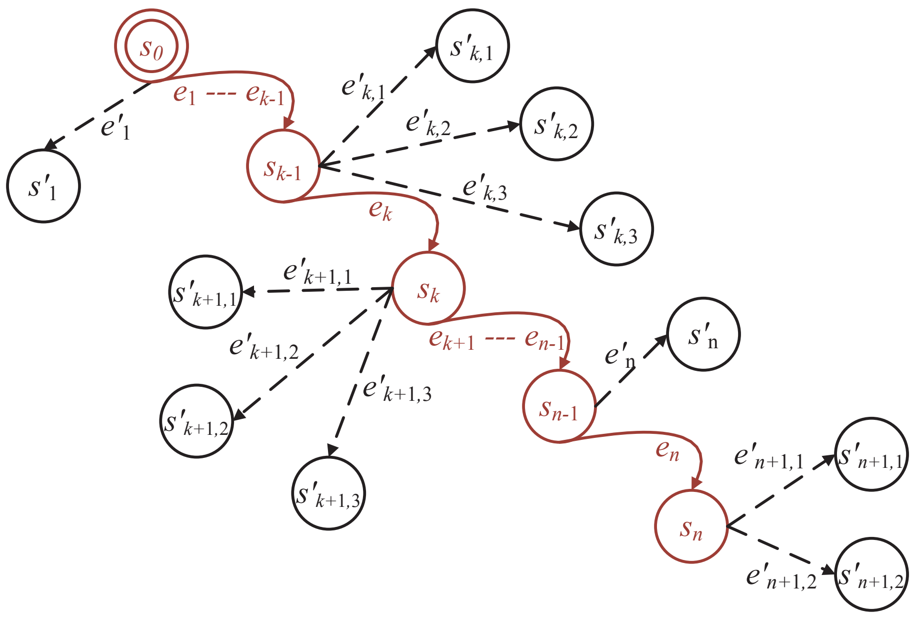


Figure 5.3: Concolic Test Generation using Virtual Devices

The virtual device starts from the initial state s_0 which is the state after resetting the device. With different subsequences of seq, the device is triggered to different states, for example, with the event sequence $seq_k = e_1, e_2, ..., e_k$, the device is brought to the device state s_k from s_0. With the set $\{s_0, \ldots, s_n\}$ of reproducible device states, we can apply symbolic execution to each of these states with a symbolic event. For each symbolic path explored, symbolic path constraints are recorded and a concrete event satisfying these constraints are generated. As shown in Figure 5.3, on state s_{k-1}, we can generate three test cases as follows:

$$\langle seq_{k-1}, e'_{k,1}\rangle,\ \langle seq_{k-1}, e'_{k,2}\rangle,\ \langle seq_{k-1}, e'_{k,3}\rangle$$

Algorithm 3 GENERATE_TEST_CASE (s_{ut}, seq, k)

```
1:  $P \leftarrow \emptyset$, $TC \leftarrow \emptyset$;
2:  $s_V \leftarrow s_{ut}$;
3:  $r_V \leftarrow Compose\_Symbolic\_Request$ ( );
4:  $d_V \leftarrow null$;
5:  $e_V \leftarrow \langle r_V, d_V \rangle$;
6:  $P \leftarrow Symbolic\_Execution$ ($s_V$, $e_V$);
7:  for each path $p \in P$ do
8:      $e \leftarrow Generate\_Concrete\_Event$ ($p$);
9:      $tc \leftarrow \langle seq_k, e \rangle$;
10:     $TC \leftarrow TC \cup \{tc\}$;
11: end for
12: return $TC$;
```

Algorithm 3 illustrates how to generate test cases. Here, P is a temporary set for saving all constraints for each path computed by symbolic execution, and TC saves all generated test cases tc. We set the given state s_{ut} as the state of virtual device s_V, and then execute the virtual device with a symbolic request r_V. In this section, we do not consider DMA data and set it to null to illustrate our algorithm. We illustrate how to handle DMA data in Section 5.2.3. For each explored path p, we can get its symbolic path constraints. Then a concrete event e is generated for triggering p. A test case tc consists of a request sequence seq_k leading the device to s_{ut} and the newly generated event e. For each s_{ut}, our approach generates a set of test cases TC.

There can be a large number of subsequences $\{seq_k\}$ in seq. To generate test cases from all $\{seq_k\}$ may entail prohibiting overheads. We allow the user to select $\{seq_k\}$ via assertions on device states and events. Then a selected seq_k is replayed on virtual prototypes to get the state under test s_{ut}. After replaying a set of selected sequences, we can obtain a set of states S_{ut} where $S_{ut} = \{s_{ut1}$, s_{ut2}, ..., $s_{utn}\}$. To help users select states more efficiently, we provide an automatic mechanism in Section 5.2.4.

5.2.3 Concolic Approach to Handling DMA Data

In order to process a device request $\boldsymbol{r}$, a device might read/write data using DMA. Therefore, we need to handle DMA data in the test generation process. We first tried a naïve approach to represent DMA data with symbolic values. We replace line 4 "$d_V \leftarrow null$;" with "$d_V \leftarrow Make\_Symbolic\_DMA$ ();" in Algorithm 3 and run the modified algorithm to generate test cases. From our experiments, we observed that it easily causes path explosion since symbolic DMA data introduces too many paths.

In order to make a virtual device work correctly, the DMA sequence for a request has to follow the device specification strictly. It is difficult to generate a whole DMA sequence using pure symbolic execution. A more promising approach is to modify DMA data in a captured DMA sequence, which means that most logic of the DMA sequence is kept. Therefore, we also utilize concolic approach to generate DMA-related test cases.

To process a concrete request r_{ut} at a concrete state s_{ut}, a concrete DMA sequence d_{ut} is accessed. We record d_{ut} in the concrete execution of a virtual device. According to Definition 4.5, $d_{ut} = d_{ut1}, d_{ut2}, ..., d_{utn}$ where d_{uti} is the ith DMA data. The length of d_{uti} is represented as l_{uti}. Usually the type of d_{uti} is a structure. An example is shown in Figure 5.4.

```
//Sample DMA data structure
struct e1000_tx_desc {
    uint64_t buffer_addr;     /* Address of the descriptor's data buffer */
    union {
        uint32_t data;
        struct {
            uint16_t length;  /* Data buffer length */
            uint8_t cso;      /* Checksum offset */
            uint8_t cmd;      /* Descriptor control */
        } flags;
    } lower;
    ......
};
```

Figure 5.4: An Example of QEMU DMA Data Structure

We further abstract the organization of d_{uti} as Figure 5.5. The data d_{uti} includes several separate sections of data, such as x, y, and z. These different sections are accessed individually to control the execution flow of the virtual device.

To collect the information of these separate sections, we first run the virtual device with r_{ut} and d_{ut} from s_{ut} concretely using the execution engine. Every time

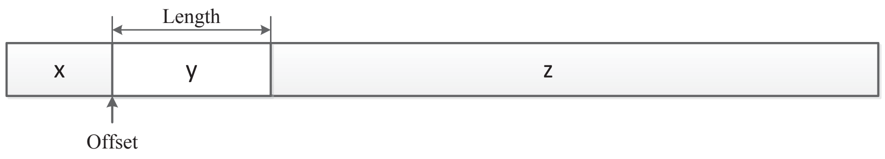


Figure 5.5: The Abstraction of a DMA Record

the DMA data is accessed, we save the offset and the length of accessed DMA data as a DMA access record a. After we finish the execution, we collect the concrete execution trace tr and get the set A of all saved DMA access records.

For each access record a in A, we define a concolic DMA data d^c_{uti} according to a as shown in Figure 5.6. The concolic DMA data d^c_{uti} includes two parts,

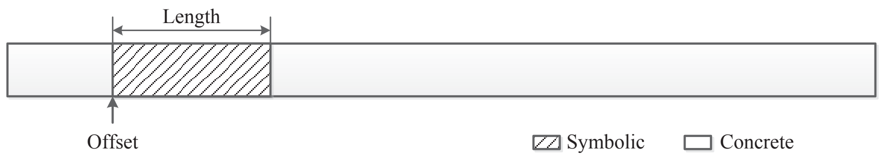


Figure 5.6: A Concolic DMA Example

a symbolic part and a concrete part. According to the offset and the length of accessed DMA data saved in a, we make symbolic that segment of d^c_{uti}, whose length is l_{uti}. We further compose the DMA sequence d^c_{ut} by combining d^c_{uti} and the rest DMA records in d_{ut}. Then we run the virtual device symbolically with d^c_{ut} and r_{ut} from s_{ut} and collect all paths explored. If only one path is explored, it means that the symbolic part does not lead to any branch forked in the symbolic execution. Therefore we do not generate a test case since the explored path follows the same trace as tr . If more than one paths are explored, for each trace different from tr, we generate a new test case based on its path constraints and d^c_{ut}.

A new algorithm as shown in Algorithm 4 is developed to generate DMA-related test cases. In Algorithm 4, P is a temporary set for saving all constraints for each

Algorithm 4 GENERATE_DMA_RELATED_TEST_CASE (s_{ut}, r_{ut}, d_{ut}, seq, k)

```
1:  $P \leftarrow \emptyset$, $A \leftarrow \emptyset$, $TC_D \leftarrow \emptyset$;
2:
3:  /* Compute the concrete trace and collect DMA access records */
4:  $s_V \leftarrow s_{ut}$;
5:  $e_V \leftarrow \langle r_{ut}, d_{ut} \rangle$;
6:  $\langle A, tr \rangle \leftarrow Concrete\_Execution\ (s_V, e_V)$;
7:
8:  /* Generate DMA-related test cases */
9:  for each access record $a \in A$ do
10:     $d_V \leftarrow Make\_Concolic\_DMA\ (d_{ut}, a)$;
11:     $e_V \leftarrow \langle r_{ut}, d_V \rangle$;
12:     $P \leftarrow Symbolic\_Execution\ (s_V, e_V)$;
13:     if $P.size() > 1$ then
14:        for each path $p \in P$ do
15:           if $Compare\_Traces(p, tr) == false$ then
16:              $d \leftarrow Generate\_Concrete\_DMA\ (p)$;
17:              $e \leftarrow \langle r_{ut}, d \rangle$;
18:              $tc \leftarrow \langle seq_k, e \rangle$;
19:              $TC_D \leftarrow TC_D \cup \{tc\}$;
20:           end if
21:        end for
22:     end if
23: end for
24: return $TC_D$;
```

path computed by symbolic execution, A is a temporary set for saving all the DMA access records, and TC_D saves all generated DMA-related test cases tc. We run a virtual device in two rounds to generate DMA-related test cases.

1. ***Compute the concrete trace and collect all DMA access records.*** We set the given state s_{ut} as the state of virtual device s_V and construct the device event e_V using the given request r_{ut} and the DMA sequence d_{ut}. Then we run the virtual device under s_V with e_V to collect the trace tr and all DMA access records A.

2. ***Generate DMA-related test cases.*** With each DMA access record a in A, we construct a concolic DMA data sequence d_V. With d_V and the concrete request r_{ut}, the virtual device is executed from s_V symbolically. All explored paths P are collected. We first determine whether P has more than one path. If it has, we compare the trace of each explored path p with tr. Then a concrete DMA sequence d is generated for triggering p if the trace of p is different from tr. A test case tc consists of a request sequence seq_k leading the device to s_{ut} and the event $\langle r_{ut}, d\rangle$.

To generate practical DMA data in the symbolic execution process, we also follow two rules. In the DMA-related test generation process, a DMA sequence is necessary for a newly explored path p. We denote such a sequence as $d = d_1, d_2, ..., d_m$ where d_i is the ith DMA data. We further denote the length of d_i as l_i. Two rules as follows are defined to guide test generation to generate well-formed DMA data.

Rule 1: If l_i is not equal to l_{uti}, p is discarded.

Rule 2: If m is larger than n, p is discarded.

In the test generation process, the length of a DMA record and the number of DMA records in a DMA sequence should be the same as the corresponding length and number of the concrete DMA data captured at runtime strictly. If there is any difference, which means there is a rule violation, the behavior of the virtual device can be very different from the concrete execution. Such kind of generated test cases are too random to trust. Our proposed rules can well eliminate these random tests.

In order to generate request-related test cases with DMA data, we extend our test case generation algorithm shown in Algorithm 3 to Algorithm 5. The extension is to use a captured or generated DMA sequence d_{ut} that is not *null* and well-formed.

Algorithm 5 GENERATE_REQUEST_RELATED_TEST_CASE (s_{ut}, d_{ut}, seq, k)

```
1:  $P \leftarrow \emptyset$, $TC_R \leftarrow \emptyset$;
2:  $s_V \leftarrow s_{ut}$;
3:  $r_V \leftarrow Compose\_Symbolic\_Request$ ( );
4:  $e_V \leftarrow \langle r_V, d_{ut} \rangle$;
5:  $P \leftarrow Symbolic\_Execution$ ($s_V$, $e_V$);
6:  for each path $p \in P$ do
7:      $r \leftarrow Generate\_Concrete\_Request$ ($p$);
8:      $e \leftarrow \langle r, d_{ut} \rangle$;
9:      $tc \leftarrow \langle seq_k, e \rangle$;
10:     $TC_R \leftarrow TC_R \cup \{tc\}$;
11: end for
12: return $TC_R$;
```

In Algorithm 5, we execute the virtual device with a symbolic request r_V and concrete DMA data d_{ut} from s_{ut}. If d_{ut} is null in the concrete execution of the virtual device, Algorithm 5 is the same as Algorithm 3. If d_{ut} is not null, we run a virtual device symbolically with a symbolic request r_V and d_{ut} from s_{ut}. The above two rules has been used in generating well-formed DMA data. They are further extended to eliminate generated requests that have DMA data ill-formed for such types of requests.

To generate request-related test cases, we not only can utilize the captured DMA sequence, but also can utilize the DMA sequences that are generated using Algorithm 4. More implementation details about utilizing DMA data are discussed in Section 3.3.3.

5.2.4 Transaction-based Test Selection Strategy

In order to make our concolic testing approach practical and efficient, we need to address the following two key challenges:

- *State selection problem.* For a virtual device, we can get a vast number of states under test by replaying a long sequence of device events. Applying test generation to all these states is impractical. How to select states under test is a critical challenge. Even if we allow users to select states with filters, it can still be a laborious process.

- *Test case redundancy problem.* Even if we only generate test cases on states selected, we can still get a large number of test cases. Applying all such test cases on a silicon device takes much time. However, certain test cases trigger the same behavior on a silicon device, i.e., they cover the same transaction. Therefore, to improve efficiency, such redundant test cases should be clearly identified.

We develop a transaction-based test selection strategy to address the above two challenges. First, states under test are selected based on device transactions. To select states, we replay a sequence *seq* of device events on the virtual device. For each state transition $s_i \overset{r_{i+1}}{\Longrightarrow} s_{i+1}$, we compute the corresponding transaction. If a new transaction t is found, we select s_i as a state under test. Based on analyzing virtual device executions in virtual machines, we observed that such states have good chances of triggering new transactions with different requests.

Algorithm 6 SELECT_STATES_UNDER_TEST (*seq*)

```
1: $StateIndices \leftarrow \emptyset$, $T \leftarrow \emptyset$;
2: $i \leftarrow 0$; //loop iteration
3: $s_0 \leftarrow Reset\_Device\ ()$;
4: while $i < seq.size$ do
5:     $e_{i+1} \leftarrow Get\_Event\ (seq, i+1)$;
6:     $s_{i+1} \leftarrow Compute\_Next\_State\ (s_i, e_{i+1})$;
7:     $t \leftarrow Compute\_Transaction\ (s_i, e_{i+1})$;
8:     if $t \notin T$ then
9:         $T \leftarrow T \cup \{t\}$;
10:        $StateIndices \leftarrow StateIndices \cup \{i+1\}$;
11:    end if
12:    $i \leftarrow i + 1$;
13: end while
14: return $StateIndices$;
```

Algorithm 6 illustrates how to select states under test in detail. *StateIndices* is a temporary set for saving indices of all selected states, and T saves all unique transactions invoked. We set the state after resetting the device as the initial state s_0. Then we run the virtual device with each event in the event sequence *seq*. After that, we compute the next state and the transaction by processing the event under the current state. For each event, if there is a new transaction t found, we save it in T and save the corresponding state index in $StateIndices$. After all events are executed, we get a set of state indices. The corresponding states are the selected states under test.

Second, we apply transaction-based test selection strategy to identify redundant test cases. In the process of selecting states as discussed above, we can get a set of unique transactions T. The set T can be further utilized to identify redundant test cases. When we conduct test generation, every time a transaction t is explored by symbolic execution, we determine whether it is a new transaction that is not in T. If it is new, the corresponding test case is saved and t is added into T. Otherwise, we save the test case as a redundant test case so that the user can utilize this test case if time permits.

5.3 IMPLEMENTATION

5.3.1 ACTG Framework

As illustrated in Figure 6.3, our automatic conconlic test generation (ACTG) framework includes three key components:

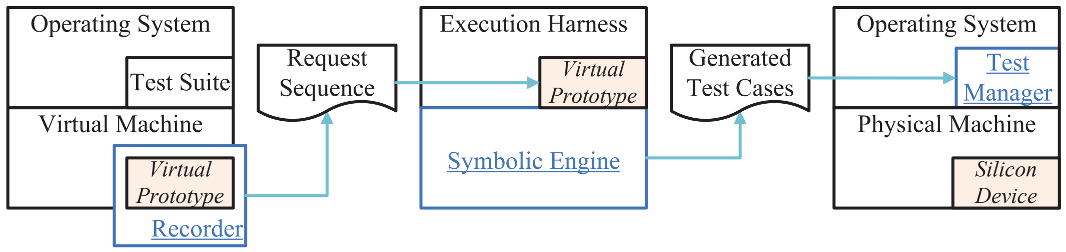


Figure 5.7: Automatic Concolic Test Generation Framework

- *Device Request Recorder.* The recorder captures device requests and DMA data from a concrete execution of the virtual device in the virtual machine. Any user or kernel level test case may be issued in the guest OS. The request recorder fully hooks the virtual device entries and DMA functions so that all device requests and DMA data are intercepted and recorded in the event sequence seq.

- *Symbolic Execution Engine.* The symbolic execution engine replays a subsequence seq_k of seq to trigger the desired state under test s_{ut}. Then the engine is used in two ways. First, the engine symbolically execute the virtual device from s_{ut} with the corresponding concrete request r_c and symbolic DMA data along the corresponding concrete trace. For each branch condition collected, a concrete DMA sequence d is generated. A new test case tc is composed of seq_k and $\langle r_c, d\rangle$. Second, the engine symbolically executes the virtual device from s_{ut} with a symbolic request. Among the transactions explored, a transaction of interest is selected, its symbolic path constraints are recorded and a concrete device request r satisfying the constraints is generated. A new test case tc is composed of the request sequence seq_k and the pair of newly generated request and the DMA sequence $\langle r, d\rangle$.

- *Test Manager.* The test manager is a kernel-level software module residing on the test machine with the silicon device. It applies a test case to the silicon device by issuing the sequence of events included in the test case.

5.3.2 Testing with Generated Test Cases

After generating test cases, our approach can then apply a generated test case to both real and virtual devices.

Application of test cases

A real (or virtual, respectively) device interacts with the high-level software in a real (or virtual) machine, on which a test case *tc* can be applied using Algorithm 7. In order to apply *tc*, we first reset both real and virtual devices so that we can keep

Algorithm 7 APPLY_TEST_CASES (TC)

```
1: for each $tc \in TC$ do
2:     $i \leftarrow 0$; //loop iteration
3:     $num \leftarrow number\_of\_requests\_in\_tc$;
4:     $s_{R,0} \leftarrow Reset\_Real\_Device$ ();
5:     $s_{V,0} \leftarrow Reset\_Virtual\_Device$ ();
6:     while $i < num$ do
7:         $e_{i+1} \leftarrow Get\_Event$ $(tc, i+1)$;
8:         $s_{R,i+1} \leftarrow Compute\_Next\_State$ $(s_{R,i}, e_{i+1})$;
9:         $s_{V,i+1} \leftarrow Compute\_Next\_State$ $(s_{V,i}, e_{i+1})$;
10:        $Check\_State$ $(s_{R,i+1}, s_{V,i+1})$;
11:        $i \leftarrow i + 1$;
12:    end while
13: end for
```

their initial states consistent. Our approach employs a test manager (a kernel-level module) to issue a *tc* in both real and virtual machines. Then we capture concrete states of both real and virtual devices after applying a *tc*. For a real device, it is difficult to capture the internal state. Hence, we only capture the interface state for the real device. Finally, we conduct consistency checks on the captured states between silicon devices and virtual prototypes. Our approach compares interface states of the real and virtual devices to detect any inconsistency. Such an inconsistency often indicates divergence between real and virtual device states, reflecting an error in either the real or virtual device.

Test case replay on virtual devices

Upon detecting an inconsistency or a hardware error, the triggering test case can be replayed on the virtual device so that the user can better understand the exercised transaction. The symbolic engine is employed for replaying a test case $tc = \langle seq, e \rangle$. The engine first brings the device to the state under test s_{ut} and then replay the event e from s_{ut}. The engine follows the same code path that it followed while generating e, since e is generated by instantiating symbolic variables to concrete values that satisfy the constraints of that path.

The power of the symbolic engine enables full controllability and observability while replaying a test case. The symbolic engine is also sufficiently responsive to support interactive replay. It enables the user to navigate backward and forward, step by step through the execution path induced by a concrete test case. Our approach can help the user better observe what variables are changed where along the path, what inputs and initial state trigger the path, and inspect values of variables at any step.

5.4 EXPERIMENTAL RESULTS

QEMU includes many virtual devices, which provides a broad range of test cases for our approach. We apply ACTG to virtual devices for three popular network adapters as shown in Table 5.1. While our tool currently focuses on QEMU-based virtual devices, the principles also apply to other virtual prototypes.

To execute virtual devices symbolically, we manually created a simple harness for each virtual device. We also created a common library of stub functions for all

Table 5.1: Virtual Prototypes for Three Network Adapters

	Vendor	Descriptions
E1000	Intel	Pro/1000 Gigabit Ethernet Adapter
Tigon3	Broadcom	BCM57xx-based Gigabit Ethernet Adapter
EEPro100	Intel	Pro/100 Ethernet Adapter

three virtual devices. The stub library has 481 lines of C code. More details about device models and their harnesses are given in Table 5.2. All device models are non-trivial in size ranging from 2099 lines to 4648 lines. All harnesses are relatively easy to create, having about 100 lines only. Only several hours are needed to create and fine-tune each harness and the stub library.

Table 5.2: Summary of Three Virtual Prototypes

	Virtual Prototype		Harness	
	Lines	Functions	Lines	Entry Functions
E1000	2099	53	74	4
Tigon3	4648	34	80	4
EEPro100	2178	70	85	7

In order to evaluate our approach, we capture a request sequence triggered by a test suite from concrete executions of virtual devices in QEMU. The test suite includes common network testing programs. As shown in Table 5.3, we give a partial list of programs in the test suite due to space limitation. For each virtual prototype, we have applied this test suite.

The experiments were performed on a desktop with an 8-core Intel(R) Xeon(R)

Table 5.3: Summary of Test Suite

Category	Commands	Descriptions
Driver Load/Unload	insmod	Load driver module
	rmmod	Remove driver module
Basic Programs	ifup	Bring a network interface up
	ifdown	Take a network interface down
	ifconfig	Configure a network interface
	ping	Send ICMP ECHO_REQUEST
	scp	Copy files between network hosts
Extra Programs	ethtool	Query or control network driver and hardware settings
	scapy	Manipulate network packets

X3470 CPU, 8 GB of RAM, 250GB and 7200RPM IDE disk drive and running the Ubuntu Linux OS with 64-bit kernel version 3.0.61.

5.4.1 Evaluation of Transaction-based Test Selection Strategy

We have applied transaction-based test selection strategy to select states and eliminate test case redundancy.

State selection

As shown in Table 5.4, we captured a large number of requests in the request sequence triggered by our test suite, for example, 64,836 requests for the E1000 virtual device. With our transaction-based test selection strategy, only a small number of states are selected, for instance, 60 states for the E1000 virtual device. In order to evaluate the efficiency of our test selection strategy, we compare it with the random strategy. With the random strategy, we select states under test randomly. Here, we select two sets of states with the random strategy. It can be observed from Table 5.4 that with the same number of states under test selected, our strategy can generate many more useful tests, i.e., tests triggering distinctive device transactions.

Table 5.4: Comparison of Different Strategies

	Requests	Transaction Strategy		Random Strategy			
	in Trace	States	Tests	States	Tests	States	Tests
E1000	64836	60	774	60	48	180	60
Tigon3	19157	52	175	52	46	156	54
EEPro100	41849	54	357	54	116	162	116

To further evaluate the efficiency of our approach, we evaluate the time usages of the transaction-based selection strategy as shown in Table 5.5. This strategy requires spending time on both selecting states and generating test cases. The overall time for E1000 is 30 minutes which includes 3.5 minutes for state selection and 26.5 minutes for test generation.

Table 5.5: Time Usage of Transaction-based Selection Strategy

	States	Time (Minutes)			
		Selection	Generation	Overall	Average
E1000	60	3.5	26.5	30	0.5
Tigon3	52	2	17	19	0.4
EEPro100	54	2	91	93	1.7

Moreover, we applied test generation to 6000 states of the E1000 virtual device selected using the random strategy. It takes 1 day, however only two new test cases are generated. If we were to apply test generation on all 64836 states, it would have taken 10 days. Through the experiment, we made two observations. First, it is not cost-effective to apply test generation to all captured virtual device states. Second, our transaction-based strategy is efficient. It brings order-of-magnitude reduction on time usage and effective and only misses a few tests found with much higher time usage.

Test case redundancy identification

As shown in Figure 5.8, our transaction-based strategy is very effective in identifying redundant tests. For each virtual device, we have achieved order-of-magnitude reduction in the number of tests that need to be applied to the virtual device in order to cover the same set of transactions. The extra tests are not thrown away and are also applied in device testing when time permits.

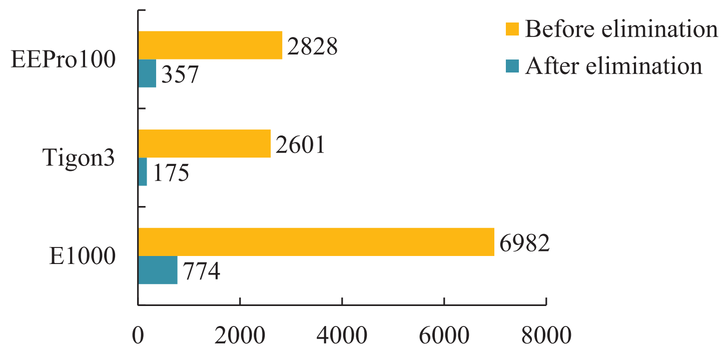


Figure 5.8: Number of Generated Tests

5.4.2 Composition of Generated Tests

We generate both DMA-related and request-related test cases. Figure 5.9 shows the number of generated DMA-related and request-related test cases on Tigon3. For Tigon3, we generate 175 test cases, 99 test cases of which are request-related test cases and 76 test cases are DMA-related test cases.

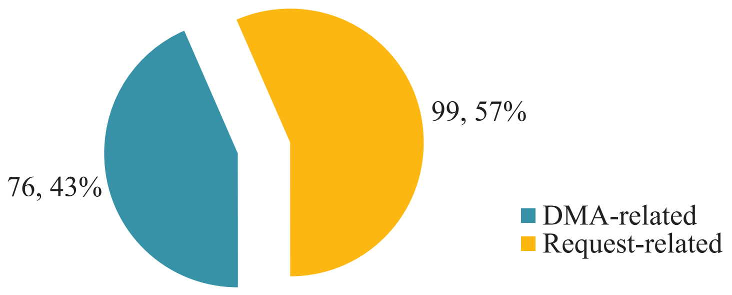


Figure 5.9: Number of Generated DMA-related and Request-related Tests

5.4.3 Evaluation of Optimization on Sparse Function Pointer Array

We present optimization results for common sparse arrays existing in a virtual device. These sparse arrays can be divided into two categories. One kind of array, denoted as A_w, includes all interface register functions for handling register-write operation; the other kind of array, denoted as A_r, includes all interface register functions for handling register-read operation.

To evaluate the sparse function pointer array optimization, we compare the number of branches forked as shown in Table 5.6. We did not conduct the evaluation on EEPro100 because EEPro100 is an old virtual device which uses several "switch-case" structures rather than the sparse function pointer array. As shown in Table 5.6, we get significant improvements using the optimization.

Table 5.6: Number of Branches Forked

	A_w		A_r	
	Without Opt.	With Opt.	Without Opt.	With Opt.
E1000	5845	13	5845	7
Tigon3	28705	13	2233	2

*Opt.: Optimization

5.4.4 Coverage Improvement

To measure the quality of generated tests, we evaluate the coverage results. We utilize test coverage over the virtual device to estimate the functional coverage over the silicon device. Because the virtual device is software, we utilize four different code coverage metrics to measure the coverage improvement.

Table 5.7: Summary of Coverage Improvement

	Statement					Block				
	#	Test Suite		Generated Tests		#	Test Suite		Generated Tests	
		#	%	#	%		#	%	#	%
E1000	3256	2602	79.91%	2835	87.07%	298	214	71.81%	252	84.56%
Tigon3	1791	1496	83.53%	1689	94.3%	138	104	75.36%	128	92.75%
EEPro100	2369	1767	74.59%	2089	88.18%	266	170	63.91%	222	83.46%

	Function					Branch				
	#	Test Suite		Generated Tests		#	Test Suite		Generated Tests	
		#	%	#	%		#	%	#	%
E1000	42	39	92.86%	42	100%	264	165	62.5%	210	79.55%
Tigon3	25	23	92%	25	100%	120	70	46.67%	97	80.83%
EEPro100	44	39	88.64%	42	95.45%	150	77	51.33%	115	76.67%

As shown in Table 6.5, the generated test cases improve test coverage significantly. Although the test suite we use has already been able to get reasonable coverage on three virtual devices, the coverage can still be significantly improved using our generated test cases. Particularly, for E1000 and Tigon3, the function coverage can be improved to 100%. For Tigon3 and EEPro100, the branch coverage can be improved by more than 25%.

5.4.5 Inconsistencies

As we apply the test cases on virtual and silicon devices, we collect both virtual and silicon device states. We then conduct consistency checking between the virtual and silicon device states. Our test cases have uncovered several inconsistencies between the real devices and their virtual devices. In our study, even though all the devices are popular devices which have gone through years of thorough testing and their virtual devices are created after fact, we still detected inconsistencies. The inconsistencies detected by our test suite and generated test cases are shown in Figure 5.10.

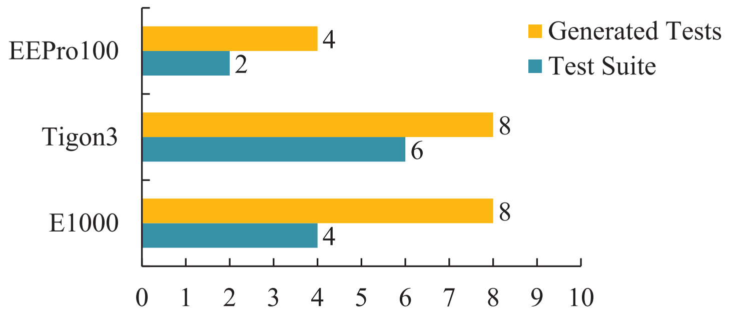


Figure 5.10: Number of Inconsistencies Detected by Test Suite and Generated Tests

One common inconsistency caused by virtual devices is that after certain special requests, one or several device registers are modified in silicon devices while in virtual devices, they are unchanged. This inconsistency was introduced assuming the drivers would well behave and not issue such special requests. Two types of inconsistencies detected are caused by silicon devices: (1) devices are not initialized properly according to device specifications and (2) devices update registers that are specified as reserved in the device specifications. We believe that if such tests are conducted on a newly designed silicon device prototype, our approach can discover more silicon device bugs.

5.5 FAULT INJECTION WITH VIRTUAL PROTOTYPES FOR DRIVER TESTING

Virtual devices are software components. Compared to their hardware counterparts, it is easier to achieve controllability on virtual prototypes. This makes virtual prototypes amenable to device fault injection for driver testing.

Based on ACTG framework and runtime shadow execution in Section 3.4, we further develop an approach to generation and injection of virtual prototype-based faults for driver testing. We first collect unique transactions to identify different device behaviors, such as register writes and interrupt firing, from concrete executions of a virtual prototype, and then modify these device behaviors to generate fault scenarios. We then employ runtime shadow execution to apply the generated fault scenarios to virtual prototypes at runtime to guide fault simulation to test whether device drivers can handle these faulty behaviors correctly. We have applied this approach to virtual prototypes of three network adaptors to generate fault scenarios. The generated faults have been applied at runtime to test driver reliability.

5.5.1 Fault Models

Before discussing the details of transaction-based fault injection, we first introduce two fault models:

- *I/O error model.* Device drivers issue requests to control and operate devices. However, these I/O requests can be lost, corrupted and bit-flipped [37, 77] due to electrical interference, bus errors and firmware failures. In our approach, we modeled two kinds of I/O faults: I/O request loss and bit-flipping.

- *Interrupt loss model.* Devices fire interrupts to notify drivers when some special events happen in the hardware. However, interrupt loss happens because of hardware failures and bus errors [37]. In our approach, we generate faults to model interrupt loss as one kind of incorrect device behaviors.

Furthermore, we generate fault scenarios for each fault model in two categories:

- *Transient Fault.* A transient fault occurs once and then disappears. In our approach, we model transient fault by simulating a fault once in one iteration of fault injection.

- *Permanent Fault.* A permanent fault is a fault that always present. In our approach, we model permanent fault by simulating a fault persistently in one iteration of fault injection.

5.5.2 Transaction-based Fault Generation

To simulate incorrect device behavior, one naïve idea is to inject faults randomly. However, there are two disadvantages with the random approach. First, incorrect device behaviors usually do not happen randomly. Second, random testing is inefficient. How to generate hardware-related and effective fault scenarios is very challenging. We further extend ACTG to generate practical and efficient fault scenarios.

Algorithm 8 TRANSACTION-BASED_FAULT_GENERATION (*seq*)

```
1:  $FaultScenarios \leftarrow \emptyset$, $T^A \leftarrow \emptyset$;
2:  $i \leftarrow 0$; //loop iteration
3:  $s_0 \leftarrow Reset\_Device$ ();
4:  while $i < seq.size$ do
5:      $e_{i+1} \leftarrow Get\_Event$ $(seq, i+1)$;
6:      $s_{i+1} \leftarrow Compute\_Next\_State$ $(s_i, e_{i+1})$;
7:      $t \leftarrow Compute\_Transaction$ $(s_i, e_{i+1})$;
8:      $addr \leftarrow Get\_Request\_Address$ $(e_{i+1})$;
9:      $t^a \leftarrow \langle t, addr \rangle$;
10:     if $t^a \notin T^A$ then
11:         $T^A \leftarrow T^A \cup t^a$;
12:         $FaultScenarios \leftarrow Generate\_Faults$ $(t^a)$;
13:     end if
14:     $i \leftarrow i + 1$;
15: end while
16: return $FaultScenarios$;
```

We first capture device requests and DMA data as a request sequence *seq* from a concrete execution of the virtual device in the virtual machine. We then employ a symbolic execution engine to replay all captured requests and generate fault scenarios using a transaction-based strategy. In the generation process, we first identify the transaction for each state transition and then use the transaction and I/O request address together as identifiers to decide whether we generate fault scenarios.

Algorithm 8 illustrates how to generate fault scenarios. $FaultScenarios$ is a temporary set for saving all generated fault scenarios, and T^A saves all unique pairs of transaction and I/O request address invoked. We set the state after resetting the device as the initial state s_0. Then we run the virtual device with each event in the event sequence seq. After that, we compute the next state and the transaction by processing the event under the current state. For each event, if there is a new pair t^a found, we save it in T^A. Based on t^a, fault scenarios are generated using fault models and saved in $FaultScenarios$. After all events are executed, we get a set of fault scenarios.

5.5.3 Fault Injection Using Runtime Shadow Execution

To apply fault scenarios, we need to detect device transaction at runtime. We employ runtime shadow execution introduced in Section 3.4 to achieve this.

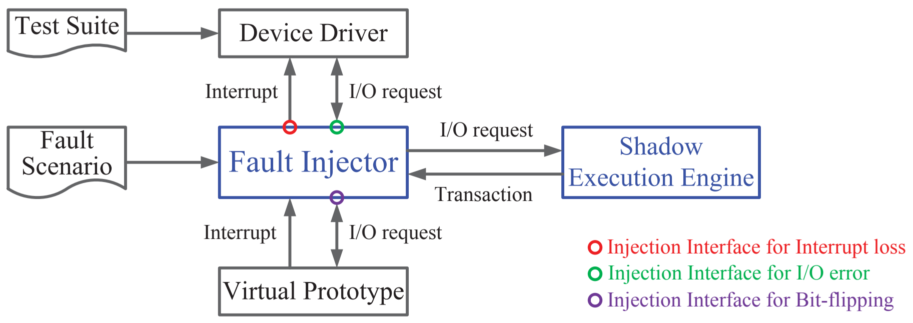


Figure 5.11: Fault Injection Framework Using Runtime Shadow Execution

Figure 5.11 shows the fault injection framework using runtime shadow execution. To apply one fault scenario, we load the fault scenario into the fault injector and then run a test suite to test device drivers. The fault injector captures each I/O request issued by the device driver, and then send I/O request information to shadow execution engine to compute the corresponding transaction. If the pair of transaction and request address matches t^a in the fault scenario, the corresponding fault model will be applied.

- *I/O request loss fault (I/O_m).* To inject an I/O request loss fault, the fault injector does not send the I/O request to the virtual prototype and return directly. This fault is injected into the I/O request interface between the driver and the fault injector.
- *I/O request bit-flipping fault (I/O_{bf}).* To inject an I/O request bit-flipping fault, the fault injector flips one bit of the I/O request following the fault scenario and sends the I/O request to the virtual prototype. This fault is injected into the I/O request interface between the fault injector and the virtual prototype.
- *Interrupt loss fault ($Intr_m$).* To inject an interrupt loss fault, the fault injector sends I/O requests to the virtual prototype and breaks the interrupt fired by the virtual prototype. This fault is injected into the interrupt interface between the driver and the fault injector.

5.5.4 Preliminary Evaluation

Using the same experimental setup in Section 5.4, we have applied our approach to virtual devices for three popular network adapters: **E1000**, **E100** and **TG3**.

As shown in Table 5.8, we have generated about 1500 fault scenarios for **E1000**, about 200 for **E100** and 2460 for **TG3**. All generation processes only take about or less than one minute. The fault injection process for each driver takes several hours. After applying all fault scenarios for three drivers, several unique warnings are triggered and 2 bugs are found on **TG3** driver. Since both **E1000** and **E100** drivers have existed for more than 15 years, it is highly possible that **E1000** and

Table 5.8: Preliminary Result of Fault Injection with Virtual Prototypes

Driver	# of faults				Time(Minutes)			# of Unique Warnings	# of Crashes
	I/O_m	I/O_{bf}	$Intr_m$	All	Generation	Injection	All		
E1000	832	674	16	1522	0.5	228.5	229	6	0
E100	100	86	6	192	1	124	125	2	0
TG3	2246	212	2	2460	0.5	620	620.5	4	2

E100 drivers can handle all kinds of device errors and not crashed. For **TG3** driver, two crashes are triggered by two different fault scenarios. One is to inject an I/O write request loss fault, the other is to inject an I/O read request loss fault. Both faults can lead to system crashes because there is no desired device behavior happening or no correct return value.

5.6 RELATED WORK

5.6.1 Symbolic execution

There has been much recent work on using symbolic execution to automatically generate test inputs, leading to software testing tools such as Java PathFinder [78], CUTE and jCUTE [73], CREST [9], BitBlaze [11], DART [30], and SAGE [32]. These tools basically follow the same approach as KLEE in solving a path's constraints to generate a test case and differ in the specifics of symbolic execution and test case generation. However, symbolic execution has a major limitation, a.k.a., path explosion. To execute a complex program symbolically, the number of feasible paths can be exponential. Furthermore, it can even be infinite in the case of programs with unbounded loop iterations [6, 52, 76].

In our approach, we applied symbolic execution to a special type of programs, virtual devices, utilized characteristics of virtual devices to improve symbolic execution effectiveness, generated test cases characterizing paths (i.e., transactions) through virtual devices, and provided facilities for applying the tests to real devices and replaying the tests on virtual devices to assist debugging.

5.6.2 Concolic testing

Concolic testing [30, 32, 73, 81], combining concrete and symbolic execution, is a hybrid software testing technique that performs symbolic execution along a concrete execution path. Concolic testing collects all path constraints along the concrete path. The path constraints are then used to incrementally generate test inputs by conjoining path constraints for a prefix of the path with the negation of a conditional taken by the execution [53]. Since the algorithm does concrete executions, all bugs inferred by the technique are real [65].

Our approach shares the general spirit of concolic testing; however, concolic execution of virtual prototypes differ significantly from previous approaches to concolic execution of software programs. In our approach, we first compute concrete device states by executing virtual prototypes with captured concrete runtime data. Then, we generate tests with concrete device states and symbolic device inputs using symbolic execution. Our approach not only integrates concrete and symbolic execution, but also combines virtual and silicon device executions.

5.6.3 Post-silicon validation

Post-silicon validation has become a bottleneck in system development cycle and is a significant, growing part of overall validation cost [38]. There has been much research on post-silicon validation to reduce costs and improve observability [22, 33, 40, 51, 63, 79]. However, many challenges remain in post-silicon validation, such as coverage metrics, failure reproduction, and test generation [57].

One approach to post-silicon test generation is Automatic Test Pattern Generation (ATPG) [24, 48], which targets exposing electrical and manufacturing defects rather than functional errors. Another approach is built-in self-test (BIST) [47, 80]. BIST is a mechanism that permits a device to test itself, which also mainly targets manufacturing defects. There has also been efforts on reusing pre-silicon validation tests in post-silicon validation [2, 60]. Our approach shares the same goal of bridging the gap between pre-silicon and post-silicon validation, while fully leveraging the white box nature of virtual prototypes to efficiently generate high-quality functional tests.

5.7 SUMMARY

This chapter presented an automatic concolic approach to generation of post-silicon tests with virtual prototypes. The generated test cases have been further issued to both virtual prototypes and silicon devices to evaluate coverage and check inconsistencies. We have obtained significant improvement in test coverage and detected 20 inconsistencies between virtual prototypes and silicon devices.

Chapter 6

AUTOMATIC DRIVER FAULT INJECTION

6.1 MOTIVATION AND OVERVIEW

Robustness testing is a crucial stage in the device driver development cycle. Device drivers may behave correctly in normal system environments, but fail to handle corner cases when experiencing system errors, such as low resource situations, PCI bus errors and DMA failures [74]. Therefore, it is critical to conduct such robustness testing to improve driver reliability. However, such corner cases are usually difficult to trigger when testing drivers. The time-to-market pressure further exacerbates the problem by limiting the time allocated for driver testing [72]. Thus, it is highly desirable to speed-up driver robustness testing and reduce human effort.

Fault injection is a technique for software robustness testing by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be traversed. Recently, fault injection techniques have been widely used for software testing [56, 61]. These techniques have major potential to play a crucial role in driver robustness testing.

In Section 5.5, we have illustrated how to conduct device fault injection for driver robustness testing. We have developed two hardware fault models to guide fault injection with virtual prototypes for driver testing. Device drivers not only interact with hardware devices, but also need kernel API support to access system resource. We have proposed a systematic fault injection approach targeting at kernel API interfaces for driver robustness testing in this chapter.

Our approach is inspired by Linux Fault Injection Infrastructure (LFII) [49] which has been integrated into the Linux kernel since Version 2.6.19. LFII can cause system faults, such as memory allocation functions returning errors, for system robustness testing. Our concept of faults is consistent with that of LFII. There are also other similar studies focusing on fault injection techniques for driver robustness testing [67, 83]. However, these approaches and tools have obvious limitations. First, they only provide basic frameworks which mainly support low memory situations. Second, they only support random fault injection which is hard to control and inefficient. Third, they require much human effort and time to get good results and are not easy-to-use. This demands an innovative approach to systematic and effective fault generation and injection for driver robustness testing.

This chapter presents an approach to automatic runtime fault generation and injection for driver robustness testing. Our approach runs a driver test and collects the corresponding runtime trace. Then we identify target functions which can fail from the captured trace, and generate effective fault scenarios on these target functions. Each generated fault scenario includes a fault configuration which is applied to guide further fault injection. Each fault scenario is applied to guide one instance of runtime fault injection and generate further fault scenarios. This process is repeated until all fault scenarios have been tested. To achieve systematic and effective fault injection, we have developed two key strategies. First, a bounded trace-based iterative generation strategy is developed for generating effective fault scenarios. Second, a permutation-based injection strategy is developed to assure the fidelity of runtime fault injection.

We have implemented our approach in a prototype driver robustness testing tool, namely, ADFI (Automatic Driver Fault Injection). ADFI has been applied to 12 widely-used device drivers. ADFI generated thousands of fault scenarios and injected them at runtime automatically. After applying all these generated fault scenarios to driver testing, ADFI detected 28 severe driver bugs. Among these bugs, 8 bugs are caused by low resource situations, 8 bugs are caused by PCI bus errors, 8 bugs are caused by DMA failures and the other 4 bugs are caused by mixed situations.

Our research makes the following three key contributions:

1)Automatic Fault Injection. Our approach to driver robustness testing not only enables runtime fault injection to simulate system errors, but also generates fault scenarios automatically based on the runtime trace to exercise possible error conditions of a driver efficiently. Our approach is easy to use and requires minimum manual efforts, which greatly reduces driver testing costs and accelerates testing process.

2)Bounded Trace-based Iterative Generation Strategy. A bounded trace-based iterative generation strategy is developed to generate unique and effective fault scenarios based on runtime traces. This strategy not only generates effective fault scenarios covering different kinds of error situations in modest time, but also produces efficient fault scenarios with no redundancy.

3)Permutation-based Replay Mechanism. To assure the fidelity of runtime fault injection with generated fault scenarios, a permutation-based replay mechanism is developed to handle software concurrency and runtime uncertainty. The mechanism guarantees that the same driver behaviors can be triggered using the same fault scenario repeatedly at runtime.

The remainder of this chapter is structured as follows. Sections 2 and 3 present the design of our approach. Section 4 discusses its implementation. Section 5 elaborates on the case studies we have conducted and discusses the experimental results. Section 6 reviews related work. Section 7 concludes our work.

6.2 BOUNDED TRACE-BASED ITERATIVE FAULT GENERATION

6.2.1 Preliminary Definitions

To help better understand our approach, we first introduce several definitions and illustrate them with examples.

*Definition 1 (**target function**):* A target function $\tilde{f}$ is a kernel API function which can fail and return an error when $\tilde{f}$ is invoked by a device driver.

As shown in Figure 2.3, the function *kmalloc* is a target function since it can fail and return a null pointer.

A stack trace records a sequence of function call frames at a certain point during the execution of a program which allows tracking the sequence of nested functions called [82].

*Definition 2 (**target stack trace**):* A target stack trace $\tau \triangleq f_1 \rightarrow f_2 \rightarrow ... \rightarrow f_n \rightarrow \tilde{f}$ of a driver consists of a sequence of driver functions and a target function $\tilde{f}$. The sequence of driver functions are called prior to $\tilde{f}$ along a driver path. The first function f_1 is a driver entry function.

A target stack trace τ records what happened before a target function was invoked. Once a driver/system crash happens, the target stack trace can help the developer better understand the driver behavior. The same target functions can appear in different target stack traces since the same target functions can be invoked along different driver paths.

As shown in Figure 6.1, when driver entry functions *Entry_A* and *Entry_B* are invoked during driver execution, there are three possible target stack traces τ_1, τ_2 and τ_3 shown in Figure 6.2.

```
void Entry_A() { //Driver entry function
    ......
    ret = Target_Function_1();
    if(!ret) goto error;
    Function_X();
    ......
}

void Function_X() {
    ......
    ret = Target_Function_2();
    ......
}

void Entry_B() { //Driver entry function
    ......
    ret = Target_Function_3();
    ......
}
```

Figure 6.1: A Driver Function Call Example

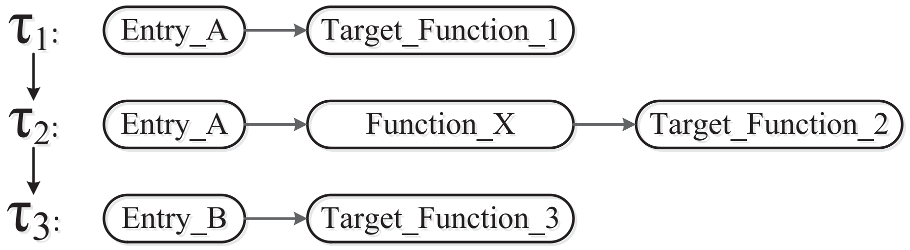


Figure 6.2: Target Stack Trace Examples

*Definition 3 (**runtime trace**):* A runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow ... \rightarrow \tau_n$ is a sequence of target stack traces. A subsequence ε_k of ε contains the first k target stack traces of ε where $\varepsilon_k \triangleq \tau_1 \rightarrow \tau_2 \rightarrow ... \rightarrow \tau_k$. A runtime trace records all target stack traces during a driver life cycle.

A runtime trace example is shown in Figure 6.2 which is $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$.

*Definition 4 (**fault configuration**):* A fault configuration $\phi \triangleq \varphi_1, \varphi_2, ..., \varphi_n$ is a sequence of boolean variables. Each boolean variable φ_i (T or F) is used for deciding whether the corresponding target function $\tilde{f}$ of τ_i invokes the kernel API or returns error. A subsequence of ϕ_k of ϕ contains the first k boolean variables of ϕ where $\phi_k \triangleq \varphi_1, \varphi_2, \varphi_3, ..., \varphi_k$.

*Definition 5 (**fault scenario**):* A fault scenario $\sigma \triangleq \langle \varepsilon, \phi \rangle$ is a pair of ε and ϕ. A fault scenario is used to guide an instance of runtime fault injection.

Suppose we capture a runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2$ and execution statuses T, T of both target fault functions in τ_1 and τ_2, then one generated fault scenario example is $\sigma \triangleq \langle \varepsilon, \phi \rangle$ where $\varepsilon \triangleq \tau_1 \rightarrow \tau_2$ and $\phi \triangleq T, F$.

*Definition 7 (**fault scenario database**):* A fault scenario database $\Sigma \triangleq \{\langle \sigma, \varsigma \rangle$ — σ is a fault scenario, ς is the fault simulation result of $\sigma\}$ is a set which saves all unique fault scenarios and their runtime execution results.

We have defined three different kinds of test results: *pass, fail and null.* Before σ is applied, ς is *null.* When the driver handles the fault scenario correctly, ς is *pass.* If the system or the driver crashes during the fault simulation, ς is *fail* and the corresponding crash report is saved for developers to conduct further analysis.

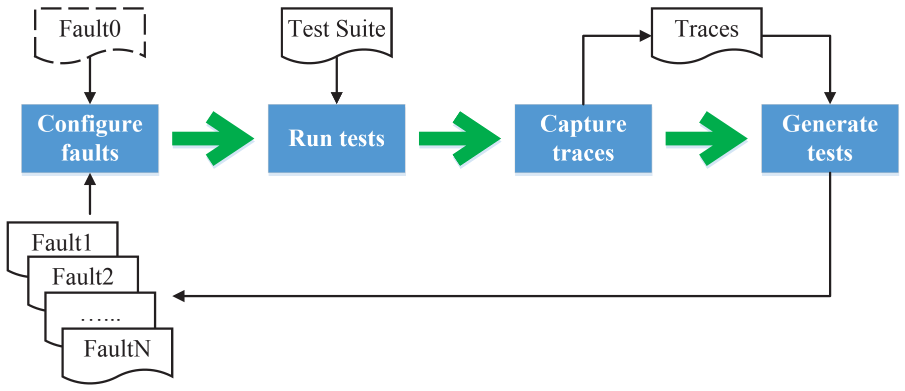


Figure 6.3: The High-level Workflow

6.2.2 Challenges

The high-level workflow of our approach is illustrated in Figure 6.3. ADFI first runs a test suite on a device driver under an empty scenario *Fault0* to capture the runtime trace where *Fault0* includes an empty configuration which does not introduce any runtime fault, and fault scenarios are generated based on the captured trace. Then given one fault scenario *FaultX*, ADFI runs the test to see if *FaultX* triggers a crash. The process of applying one fault scenario is one instance of runtime fault injection. In one instance, ADFI hooks all target function calls. Each time a target function call is captured, ADFI decides to execute the corresponding target function or inject a fault (return a false result) according to the fault scenario. Simultaneously, ADFI collects the trace executed during this run. Next, ADFI generates more fault scenarios based on the trace. The above process is repeated until all fault scenarios are applied.

The approach described above has two major challenges.

Fault scenario explosion: Generating all feasible fault scenarios does not scale if a large number of target functions exist in a driver. A naïve approach to generating fault scenarios is to explore all target function combinations along a driver runtime trace ε. If there are N target functions along ε, the number of generated fault scenarios can be $2^N - 1$. If we apply all these fault scenarios to driver robustness testing, it can take much time or even forever. Indeed as we tried this approach, it caused a fault scenario explosion after applying a few fault scenarios.

Handling concurrency and runtime uncertainty: ADFI repeatedly runs the same test suite and applies different fault scenarios to guide runtime fault injection. A fault scenario σ is a pair of a reference runtime trace ε and a fault configuration ϕ. To apply σ, ADFI captures a new runtime trace ε_{new} and run each target function $\varepsilon_{new}.\tau_i.\tilde{f}$ according to ϕ. Due to system concurrency and runtime uncertainty, ε and ε_{new} can be different which brings difficulty to find the right $\phi.\varphi_i$ to guide fault injection. This demands a systematic replay mechanism to guarantee that ε_{new} conforms to ε upon a given fault configuration ϕ.

6.2.3 Trace-based Iterative Strategy

In order to address the fault scenario explosion challenge, we have developed a bounded trace-based iterative generation strategy. For each fault scenario σ, ADFI runs the test suite on the driver and captures the runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow$ ...$\rightarrow \tau_n$. In the following, we set n to 3 to illustrate our approach. Although we use a small number as the example, the idea can be applied to any large number. As shown in Figure 6.4(a), we capture a runtime trace which includes three stack traces and the corresponding execution statuses of target functions in three stack traces: (T, T, T).

By applying the naïve approach, we can generate seven $(2^3 - 1)$ fault scenarios. However, some generated fault scenarios are invalid fault scenarios which are not feasible at runtime. For example, if a generated fault configuration $\phi \triangleq (T, F)$

is applied, the actual trace is $\tau_1 \to \tau_2 \to \tau_4$ shown in Figure 6.4(c) which is different from $\tau_1 \to \tau_2 \to \tau_3$. In this case, (T, F, F) would be an invalid fault configuration for the trace $\tau_1 \to \tau_2 \to \tau_3$. In order to avoid generating invalid fault scenarios, our trace-based iterative generation strategy only generates one-step fault configurations (F), (T, F) and (T, T, F) in this iteration as shown in Figure 6.4(b).

Remark: Our approach does not miss any valid fault scenarios. If the driver works as shown in Figure 6.4(a), our trace-based iterative generation strategy first generates three fault scenarios. Then after the fault scenario including the configuration (F) is applied, the captured fault trace should be (F, T, T) and we can generate new fault configurations (F, F) and (F, T, F). After we apply all fault scenarios, we can cover all eight possibilities eventually.

Moreover, our trace-based iterative generation strategy only generates new fault scenarios on a newly captured stack trace. Suppose we apply fault configuration (T, F) generated in Figure 6.4(b), we can capture the runtime trace $\tau_1 \to \tau_2 \to \tau_4$. As shown in Figure 6.4(c), we only generate one new fault configuration (T, F, F) from the captured target function execution trace (T, F, T). Here, we do not generate a fault configuration (T, T) because it has been covered. In this way, no duplicate fault scenarios (configurations) are generated.

Algorithm 9 illustrates how to generate new fault scenarios using the trace-based iterative generation strategy. The algorithm takes a runtime trace ε, a reference fault scenario σ and the fault scenario database Σ as inputs. If the length of the configuration is less than the length of ε, the algorithm first supplements the configuration by adding $(j - i)$ true decisions into ϕ to build a complete configuration (line 2). The algorithm goes through subsequences of the runtime trace ε between ε_j and ε_i. For each subsequence ε_i, the algorithm constructs a new fault

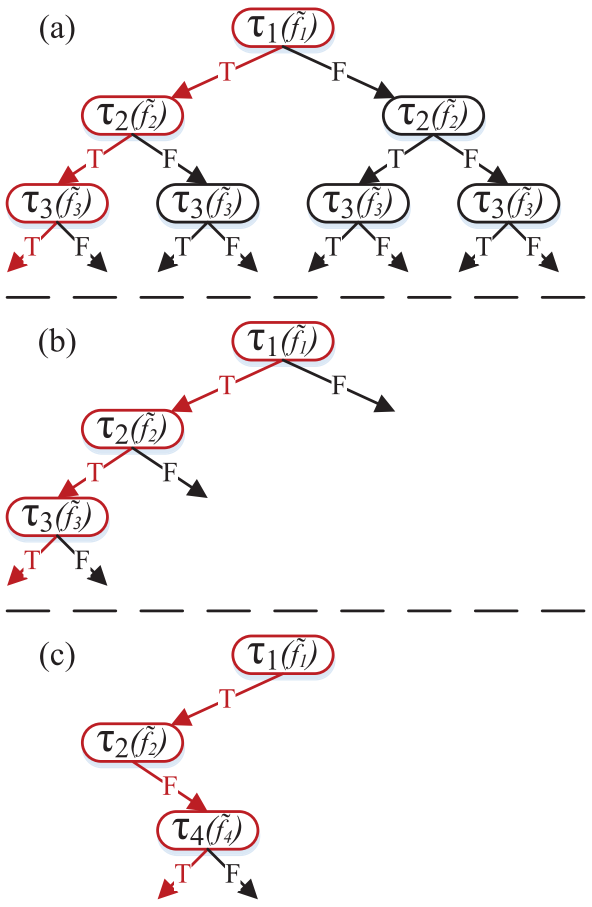


Figure 6.4: Trace-based Iterative Generation Example

Algorithm 9 ITERATIVE_GENERATION (ε, σ, Σ)

```
1: $i \leftarrow \varepsilon.size();\ j \leftarrow \sigma.size();\ \phi \leftarrow \emptyset;$
2: $\phi \leftarrow buildCompleteConfiguration(\sigma.\phi,\ i,\ j);$
3: while $i > j$ do
4:     $\phi_{new} \leftarrow \phi_j, 0;$ //Build a new configuration
5:     $\Sigma.insert(\langle \varepsilon_{j+1},\ \phi_{new} \rangle);$ //Save the fault scenario
6:     $j \leftarrow j + 1;$
7: end while
```

decision ϕ_{new} by combining the subsequence ϕ_{i-1} of the previous fault decision ϕ and a false decision. A new fault scenario is created which includes ε_i and ϕ_{new} and saved into the database Σ. Suppose we apply a fault configuration $\phi \triangleq (T, F)$ and capture the corresponding runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_4$, the corresponding length i is 3 and j is 2. We first supplement the configuration as $\phi \triangleq (T, F, T)$, then we build a new configuration $\phi \triangleq (T, F, F)$.

6.2.4 Bounded Generation Strategy

We have applied the trace-based iterative generation strategy to device drivers and it can greatly reduce the number of generated tests. However, there are still a large number of fault scenarios generated. After analyzing the captured runtime trace, we found that there are two main reasons.

1)Duplicate stack traces. For some drivers, many duplicate stack traces exist in a runtime trace. There are mainly two reasons for duplicate stack traces. First, the same target function is repeatedly invoked within a loop. For example, a set of ring buffers is usually allocated using a loop when a network driver is initialized. Second, the same target function is invoked along a driver path and the driver path is frequently executed for processing special requests. For example, system resources are allocated and freed in the transmit function for a network driver and the transmit function is called many times during an instance of driver testing.

2)Fault scenario explosion. Although we have applied the trace-based iterative generation strategy to eliminate invalid fault scenarios, fault scenario explosion still exists. As shown in Figure 6.4(a), eight fault scenarios can be all valid for some drivers. If there are N target functions along a runtime trace, a subset of all N target functions (the number is M, $M < N$) can still bring a large amount of fault scenarios (the number can be $2^M - 1$) in the final result.

To solve these two problems, we have developed a bounded generation strategy to avoid injecting an exponential number of fault scenarios. ADFI supports two kinds of bounds: maximum number of injected faults on the same stack traces in a fault scenario (*MSF*) and maximum number of injected faults in a fault scenario (*MF*).

First we explain how *MSF* works. Suppose *MSF* is 1, we use an example to illustrate the idea. We captured a runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ and the corresponding target function execution trace (F, T, T). Within ε, τ_1 and τ_3 are the same stack traces. If we generate fault scenarios following the trace-based iterative strategy, we should generate two fault configurations (F, F) and (F, T, F). The bounded generation strategy does not allow us to inject more than one fault on the same stack trace, which means we only generate one fault configuration (F, F). For another bound *MF*, the idea is straightforward. The number of injected faults in a fault scenario cannot exceed *MF*.

As shown in Algorithm 10, we have extended Algorithm 9 to support bounded generation. There are mainly three differences. First, we go through the reference fault scenario σ to record all fault-related stack traces and the number of faults as a map T before generating tests. Second, before fault scenarios are generated, we check whether the number of faults in the reference fault scenario exceeds *MF*. If yes, we terminate test generation and return directly. Third, during the generation, we check whether the number of faults injected on the same stack traces exceeds *MSF*. If not, we generate the corresponding fault scenario. Otherwise, no fault scenario is generated.

As shown in Algorithm 11, we process the fault scenario σ to record all fault-related stack traces. $T \triangleq \{\langle \tau, count \rangle$ — τ is a stack trace, *count* is the number of

Algorithm 10 BOUNDED_GENERATION (ε, σ, Σ, *bound*)

```
1: $i \leftarrow \varepsilon.size();\ j \leftarrow \sigma.size();\ \phi \leftarrow \emptyset;\ T \leftarrow \emptyset$
2: $\phi \leftarrow buildCompleteConfiguration(\sigma.\phi,\ i,\ j)$;
3: $T \leftarrow recordAllFaults(\sigma)$;
4: if $checkMFBound(T,\ MF)$ then
5:    return
6: end if
7: while $i > j$ do
8:    if $checkMSFBound(T,\ \varepsilon.\tau_{j+1},\ MSF)$ then
9:       $\phi_{new} \leftarrow \phi_j, 0$; //Build a new configuration
10:      $\Sigma.insert(\langle \varepsilon_{j+1},\ \phi_{new} \rangle)$; //Save the fault scenario
11:   end if
12:   $j \leftarrow j + 1$;
13: end while
```

Algorithm 11 RECORDALLFAULTS (σ)

```
1: $\varepsilon \leftarrow \sigma.\varepsilon$; $\phi \leftarrow \sigma.\phi$; $T \leftarrow \emptyset$; $i \leftarrow \sigma.size()$; $j \leftarrow 1$;
2: while $i \geq j$ do
3:     if $\phi.\varphi_j == F$ then
4:         if $T.find(\varepsilon.\tau_j) == T.end()$ then
5:             $T.insert(\varepsilon.\tau_j, 1)$;
6:         else
7:             $T.find(\varepsilon.\tau_j) \leftarrow T.find(\varepsilon.\tau_j) + 1$;
8:         end if
9:     end if
10:    $j \leftarrow j + 1$;
11: end while
12: return $T$;
```

faults injected on $\tau\}$ is a map. We process each boolean variable $\phi.\varphi_j$ in the fault configuration. Once $\phi.\varphi_j$ is false, we insert $\langle \varepsilon.\tau_j, 1\rangle$ into T or increase the *count* by 1 if $\varepsilon.\tau_j$ exists in T.

6.3 PERMUTATION-BASED INJECTION STRATEGY

Even if we issue the same test suite to device drivers, two runtime traces ε_1 and ε_2 can be different due to driver concurrency, runtime uncertainty, such as timing issues, memory allocation status and network overload.

There are three kinds of possible differences between ε_1 and ε_2 triggered by the same test suite.

1)Different sequences of stack traces. Device drivers are system software which can handle more than one requests at the same time, which means concurrency widely exists in device drivers. Due to the concurrency, even if two captured runtime traces include the same stack traces, the sequence of stack traces can be different between ε_1 and ε_2.

2)Different length of runtime traces. Due to different system situations or environments, the number of the same stack trace τ can be different between ε_1 and ε_2. For example, if we send the same data over a network driver, there can be different number of calls to the transmit function of the driver. This difference brings different number of the same τ existing in ε_1 and ε_2.

3)Different number of unique stack traces. Due to different faults injected, stack traces captured can be different between ε_1 and ε_2. Since fault scenarios trigger different driver paths, ε_1 and ε_2 along different paths can include different stack traces.

Since a fault scenario σ is generated based on a runtime trace, there are the same differences between $\sigma.\varepsilon$ and the corresponding triggered runtime trace ε_{new}. This makes it difficult to guide runtime fault scenario injection.

We first illustrate how to resolve the first difference. A fault scenario σ includes a runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow ... \rightarrow \tau_n$ and a fault configuration $\phi \triangleq \varphi_1, \varphi_2, ..., \varphi_n$. To guide fault injection at runtime, it might trigger a new runtime trace $\varepsilon_{new} \triangleq \tau_{new_1} \rightarrow \tau_{new_2} \rightarrow ... \rightarrow \tau_{new_n}$. Here we assume that ε and ε_{new} have the same stack traces, later we will illustrate how to handle different stack traces. ε should be a permutation of ε_{new}, which means ε_{new} is constructed by all stack traces in ε with a different sequence. As an example, $\tau_1 \rightarrow \tau_2 \rightarrow \tau_4 \rightarrow \tau_3$ is a permutation of $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$. In the runtime fault injection, we detect such permutations automatically and guide the fault injection.

The second difference is caused by runtime uncertainty. Here we assume that

ε and ε_{new} include the same set of unique stack traces and the lengths of ε and ε_{new} can be different, later we will discuss how to handle different set of unique stack traces. Based on the analysis of driver code and our observation, repeatedly injecting faults on the same stack traces caused by runtime uncertainty does not trigger new bugs. Therefore we just ignore such kinds of differences.

Algorithm 12 GET_FAULT_CONFIGURATION (τ, σ, $Flags$)

```
1: $i \leftarrow 0;\ n \leftarrow Flags.size();$
2: $i \leftarrow findNextStackTrace(\tau, \sigma, i);$
3: while $i \neq n$ do
4:   if $Flags[i] \neq true$ then
5:     $Flags[i] \leftarrow true;$
6:     return $\sigma.\phi.\varphi_i;$
7:   end if
8:   $i \leftarrow findNextStackTrace(\tau, \sigma, i);$
9: end while
10: return $true;$
```

As shown in Algorithm 12, a permutation-based injection mechanism is developed to guide the fault configuration. The algorithm takes a stack trace τ, the fault scenario σ and a flag array $Flags$ as inputs. The array $Flags$ has the length of $\sigma.\varepsilon$ and each element is initialized as false at the beginning of an instance of fault injection. Each time a target function is invoked, we determine whether the function should be executed normally or return an error with the corresponding stack trace τ. We first find τ from the beginning of σ and return the index i. Then we check $Flags[i]$ to see whether the fault decision $\sigma.\phi.\varphi_i$ has been conducted or not. If it is not conducted, we return $\sigma.\phi.\varphi_i$. Otherwise, we continue to get the index of the next stack trace from the position i. If we can not get the index from a position, $findNextStackTrace$ function returns n which means all fault decisions for τ have been covered. Therefore we return true to let the target function execute normally.

The third difference is caused by different faults injected. A set of unique stack traces in ε and ε_{new} is represented as S_ε and $S_{\varepsilon_{new}}$. There can be three kinds of cases: $S_\varepsilon \subsetneq S_{\varepsilon_{new}}$, $S_{\varepsilon_{new}} \subsetneq S_\varepsilon$ and $(S_\varepsilon \nsubseteq S_{\varepsilon_{new}}\ and\ S_{\varepsilon_{new}} \nsubseteq S_\varepsilon)$. According to our experiments, only the first case $S_\varepsilon \subsetneq S_{\varepsilon_{new}}$ occurs. There are two reasons. First, the same test suite is used for different rounds of fault injections. Second, a fault injected can trigger some new stack traces. Currently we also detect two other kinds of cases in our tool. Once any case is found, a warning is given.

6.4 IMPLEMENTATION

6.4.1 Overview

As illustrated in Figure 6.5, our automatic fault injection framework includes three key components:

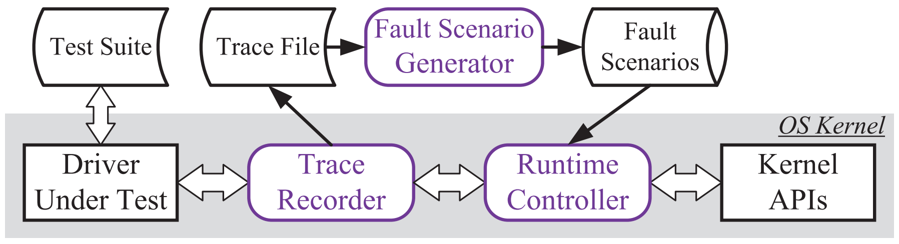


Figure 6.5: Runtime Fault Injection Framework

1) Trace Recorder. The trace recorder captures runtime traces and kernel function return values while the driver is tested under a test suite. The trace recorder fully hooks the kernel API function calls so that all function calls and return values are intercepted and recorded in the trace files.

2)Fault Scenario Generator. The fault scenario generator takes a trace file as the input to generate fault scenarios. A trace-based iterative generation algorithm is implemented and employed by the generator to deliver high-quality fault scenarios. Generated fault scenarios are saved in the fault scenario database for guiding further fault injection.

3)Runtime Controller. The runtime controller applies a fault scenario in the driver testing process by emulating a fault return according to the fault configuration. The runtime controller is a kernel-level module working with the trace recorder together. It intercepts all target function calls invoked by device drivers. Once a kernel API function call is captured, it determines if a fault should be injected. If it is, the runtime controller returns a false result instead of invoking the real kernel API function.

6.4.2 Fault Injection on Kernel API Interface

In this dissertation, we mainly focus on the kernel API functions provided by the kernel since we want to test whether device drivers can survive under different system situations. Since operating systems provide lots of kernel API functions to support drivers, so far we have conducted our research on three main categories of kernel API functions:

1)Memory Allocation Functions. The Linux kernel offers a rich set of memory allocation primitives which can be used by device drivers to allocate and optimize system memory resources. Different kinds of memory allocation functions can be used for allocating different kinds of memory. For example, the "*kmalloc*" function is used to grab small pieces of memory in kernel space and "*get_free_page(s)*" function is used to allocate larger contiguous blocks of memory.

2)Memory Map and DMA Functions. A modern operating system is usually a virtual memory system, which means that the addresses seen by user programs do not directly correspond to the physical address used by the hardware devices. Memory map functions are needed for the conversion between virtual address and physical address. For example, the "*mmap*" function establishes a mapping between a process address space and a device. DMA is the hardware mechanism used for data transfer between device drivers and hardware devices without the need of involving the system processor. For example, the the "*dma_set_mask*" function is used for checking if the mask is possible and updates the device parameters if it is.

3)PCI Interface Functions. PCI bus is a widely-used system bus for attaching hardware devices. To support PCI device control and management, a set of functions are provided by the kernel and used by device drivers. For example, the "*pci_enable_device*" function is used for initializing device before it is used by a driver.

6.4.3 Filter Mechanism

When we first applied ADFI, we observed that the same crashes happened repeatedly. After analyzing these crashes, we found two key reasons.

1)Caused by a target function. If a fault is injected into a target function $\tilde{f}$, the corresponding error handling code for $\tilde{f}$ is tested. If the error handling mechanism is not correct, there is always a crash if a fault is injected on $\tilde{f}$ in a fault scenario.

2)Caused by a sequence of stack traces. Suppose a fault scenario σ_1 includes a runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$ and a fault configuration $\phi \triangleq T, F, T, F$, it triggers a crash. If another fault scenario σ_2 includes the same runtime trace and a different fault configuration $\phi \triangleq F, F, T, F$, σ_2 possibly causes the same crash. In σ_1, two faults are injected in τ_2 and τ_4 which cause a crash. Since the same two faults are injected in τ_2 and τ_4 within σ_2, the same crash usually happens according to our experiments.

The target function $\tilde{f}$ is included in different stack traces. The stack trace τ is included in different fault scenarios. If we detect a bug triggered by a specific target function or a stack trace or a sequence of stack traces, we do not want to trigger the same crash repeatedly by other fault scenarios. Currently we provide two kinds of filter mechanisms to avoid such kinds of repeated crashes.

1)Function-Call-based Filter. A function call can be labeled as a filter pattern. As long as a fault needs to be injected into this function call according to the fault configuration, the fault scenario is ignored and not applied.

2)Stack-Trace-based Filter. A stack trace (or a sequence of stack traces) can be defined as a filter pattern. As long as a fault (or a sequence of faults) needs to be injected into a stack trace (or a sequence of stack traces, respectively) according to the fault configuration, the fault scenario is ignored and not applied.

The filter mechanism provides flexibility for driver developers to define filters to avoid repeated crashes. It has been applied to both fault scenario generation and injection.

6.5 EXPERIMENTAL RESULTS

6.5.1 Experimental Setup

As shown in Table 6.1, we applied ADFI to 12 drivers in 3 categories: Wireless, USB controller and Ethernet. These three categories represent the most important three types of PCI devices.

As the workloads of the experiments, we created different test suites for different

Table 6.1: Summary of Target Drivers

Category	***Driver***	***Size***	***Description***
Wireless	ath9k	4.3M	Qualcomm AR9485 Wireless Driver
	iwlwifi	12M	Intel Wireless AGN Driver
USB	ehci_hcd	10M	USB 2.0 Host Controller Driver
	xhci_hcd	13M	USB 3.0 Host Controller Driver
Ethernet	e100	655K	Intel(R) PRO/100 Network Driver
	e1000	2.3M	Intel(R) PRO/1000 Network Driver
	ixgbe	5.9M	Intel(R) 10 Gigabit Network Driver
	i40e	8M	Intel(R) 40 Gigabit Network Driver
	tg3	2.1M	Broadcom Tigon3 Ethernet Driver
	bnx2	1.3M	Broadcom NetXtreme II Driver
	8139cp	537K	RealTek Fast Ethernet driver
	r8169	1.1M	RealTek Gigabit Ethernet Driver

categories. There is one requirement that each test suite must start with a "load driver" command and end with a "remove driver" command. Between them, any test cases are allowed. A partial list of test cases for each category is shown in Table 6.2. Of these drivers, Intel ethernet network drivers are downloaded[1]. The other drivers are from Linux kernel source code.

6.5.2 Bug Findings

After testing all 12 drivers, we found the 28 distinct bugs described in Table 6.3. Of these bugs, 8 bugs are triggered by PCI interface faults, 8 bugs are triggered by memory allocation faults, 8 bugs are triggered by DMA function faults, and the other 4 bugs are triggered by mixed PCI/Memory/DMA faults. All these bugs can result in serious driver/system issues which include driver freeze, driver crash,

[1]The latest version of Intel ethernet network drivers can be download in the following link: http://sourceforge.net/projects/e1000/files/

Table 6.2: Summary of Workload

Category	***Test Applications***
Wireless	Basic network commands (e.g. *ifup*, *ifconfig*, *ifdown*)
	Data transfer commands (e.g. *scp*, *ping*)
	Wireless config tools (e.g. *iw*, *iwconfig*)
USB	Basic USB control commands (e.g. *lsusb*)
	Enable/disable a USB device on the USB hub
	Transfer data to a USB disk
Ethernet	Basic network commands (e.g. *ifup*, *ifconfig*, *ifdown*)
	Data transfer commands (e.g. *scp*, *ping*)
	Ethernet config tools (e.g. *ethtool*, *scapy*)

Table 6.3: Bug Results

Category	Wireless Driver		USB Driver		Ethernet Driver								Total
	ath9k	iwlwifi	ehci_hcd	xhci_hcd	e100	e1000	ixgbe	i40e	tg3	bnx2	8139cp	r8169	
PCI	0	0	0	0	0	0	2	2	0	2	2	0	8
Memory	0	1	0	0	1	4	0	2	0	0	0	0	8
DMA	1	0	0	4	1	0	0	0	0	1	1	0	8
Mixed	0	0	0	0	0	4	0	0	0	0	0	0	4
Total	1	1	0	4	2	8	2	4	0	3	3	0	***28***

system freeze and system crash. Moreover, all these bugs are difficult to find under normal situations.

These results show the effectiveness of our fault injection approach. We summarize the failure outcomes as follows:

1)System crash. The fault results in a kernel panic or fatal system error which crashes the entire system.

2)System hang. The fault results in a kernel freeze where the whole system ceases to respond to inputs.

3)Driver crash. The fault only results in a driver crash while the system can still work correctly.

4)Driver freeze. The fault only results in a driver freeze where the driver can not be loaded/removed.

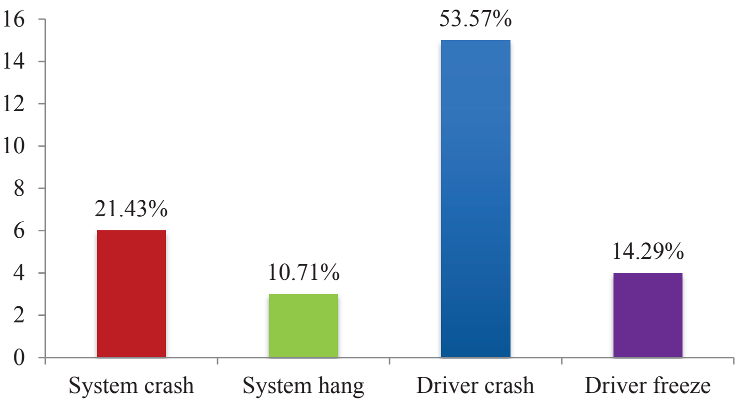


Figure 6.6: Outcomes of Experiments

Figure 6.6 provides the distributions of failure types. Of the 28 bugs, 9 bugs result in system failures including 6 system crashes and 3 system hangs. The other 19 bugs result in driver failures including 15 driver crashes and 4 driver freezes.

Bug Validation. To verify if all these bugs are valid, we manually injected bug-correlated faults into device drivers. For example, if there is a "*kmalloc*" fault, we manually injected the fault. We modified the original statement

*"void * p = kmalloc(size, GFP_KERNEL);"*

to

*"void * p = NULL;"*

Then we recompiled the driver and ran the driver under the test suite. The above example is just a simple fault scenario. Some fault scenarios are quite involved and require more modifications to the driver code to reproduce. All 28 bugs can be triggered the same way as they are triggered by ADFI. By this manual validation, we are better assured that all 28 bugs are valid and they can happen in a real system environment.

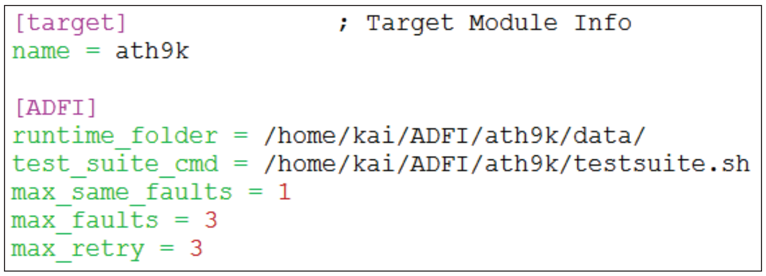
6.5.3 Human Efforts

One goal of ADFI is to minimize the human effort in testing the robustness of a driver. The necessary effort of our approach comes from three sources: (1) a configuration file to prepare ADFI for testing a driver; (2) crash analysis; (3) compilation flag modification to support coverage. The first two efforts are required for our approach while the third one is optional.

Configuration file. Only a few parameters need to be defined in a configuration file. They include driver name, runtime data folder path, test suite path and several runtime parameters. One example is shown in Figure 6.7. Such configuration is easy to create. In our experiments, only a few minutes are needed to set up one configuration file.

Table 6.4: Results under Different MF (MSF = 1)

Category	MF	Wireless Driver		USB Driver		Ethernet Driver							
		ath9k	iwlwifi	ehci_hcd	xhci_hcd	e100	e1000	ixgbe	i40e	tg3	bnx2	8139cp	r8169
PCI	1	1	3	0	0	2	5	5	5	7	3	2	2
	2	1	3	0	0	2	9	8	8	10	3	2	3
	3	1	3	0	0	2	9	9	8	10	3	2	3
Memory	1	5	24	4	1	3	13	11	32	11	9	1	3
	2	5	164	10	1	3	49	53	136	25	34	1	3
	3	5	840	12	1	3	117	156	414	29	51	1	3
DMA	1	3	4	1	6	5	11	9	17	4	13	3	8
	2	3	9	1	6	6	40	51	69	6	77	7	24
	3	3	10	1	6	6	95	171	177	6	221	8	37
ALL	1	9	31	5	7	10	28	25	54	22	25	6	13
	2	9	235	15	7	12	180	209	268	84	234	10	56
	3	9	1375	18	7	12	858	924	1365	175	980	11	130



```
[target]                ; Target Module Info
name = ath9k

[ADFI]
runtime_folder = /home/kai/ADFI/ath9k/data/
test_suite_cmd = /home/kai/ADFI/ath9k/testsuite.sh
max_same_faults = 1
max_faults = 3
max_retry = 3
```

Figure 6.7: A Sample Configuration

Crash analysis. Once a crash happens, the developer needs to figure out the cause of the crash. Our approach can inject the same fault and trigger the same behavior repeatedly. When there is a crash, our approach can tell what faults have been injected into the driver. Furthermore, the whole driver stack is provided by ADFI to support crash analysis. This information can help driver developers understand and figure out the root cause of the crash. In our experiments, the average time for understanding each of the 28 bugs is less than 10 minutes using the ADFI debug facilities.

Compilation flag. In order to evaluate the driver code coverage, we need to compile the driver with additional compilation flags. We can achieve this in two ways. First, we can add the flags into the Linux kernel compilation process. Second, we can add the flags into the driver compilation Makefile. Both ways are easy to implement. In our experiments, we manually added the compilation flags into each driver Makefile.

6.5.4 Evaluation of Fault Generation and Injection Strategy

ADFI allows two kinds of bounds, the maximum faults (MF) and the maximum same faults (MSF) in a test case. We first set MSF as 1 and then generated faults under different MFs. Table 6.4 shows the number of generated fault scenarios where MF is 1, 2 and 3.

We have generated fault scenarios on all functions in the three categories (c.f. Section 5.2). As shown in Table 6.4, different number of fault scenarios were generated for different device drivers. For drivers such as *ath9k* and *8139cp*, only about 10 fault scenarios were generated. For drivers such as *iwlwifi* and *i40e*, more than 1000 fault scenarios were generated. The number of generated faults depends on how many target functions are used in a device driver.

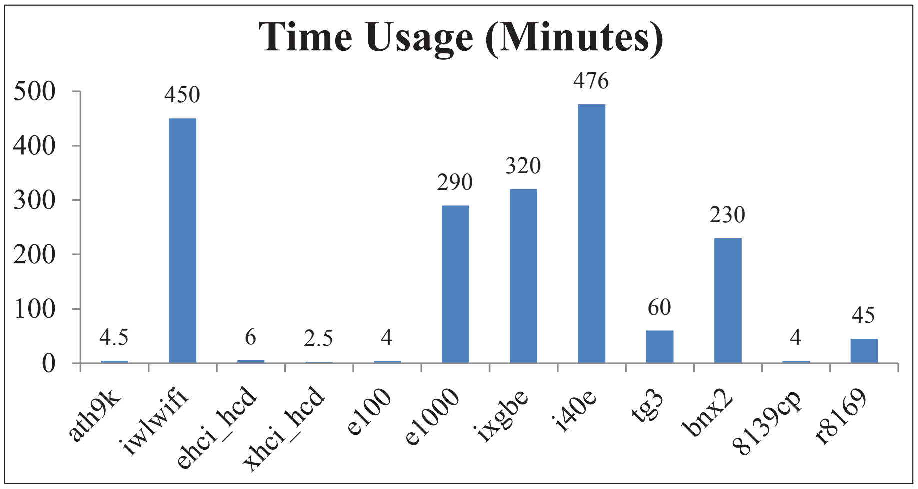


Figure 6.8: Time Usage

Another observation from the results is that there are no generated fault scenarios for *ehci_hcd* and *xhci_hcd* under PCI category. After analyzing the source code of *ehci_hcd* and *xhci_hcd* code, we did not find PCI-related functions invoked by these drivers directly. The fact is that both these drivers only invoke some PCI wrapper functions directly and these PCI wrapper functions are defined in the kernel.

We further tried to generate fault scenarios while setting MSF as 2 on *e1000* and *iwlwifi* drivers. We generated more test cases on both drivers, however no new bugs were detected and almost no coverage improvement was achieved.

In order to evaluate the efficiency of ADFI, we summarized total time usage for fault generation and injection in Figure 6.8. All these time usages were summarized while generating fault scenarios on all functions in three categories. ADFI can deliver high quality fault scenarios and find bugs effectively with a modest amount of time.

Table 6.5: Summary of Coverage Improvement

Driver	Statement					Branch				
	#	Test Suite		Generated Tests		#	Test Suite		Generated Tests	
		#	%	#	%		#	%	#	%
ath9k	6146	3147	51.20%	3208	52.20%	3171	1059	33.40%	1268	39.99%
iwlwifi	11966	6761	56.50%	7000	58.50%	6458	2454	38.00%	2648	41.00%
ehci_hcd	2763	1307	47.30%	1323	47.88%	1586	568	35.81%	588	37.07%
xhci_hcd	4772	2114	44.30%	2119	44.40%	2485	721	29.01%	723	29.09%
e100	1258	721	57.31%	743	59.06%	617	206	33.39%	231	37.44%
e1000	5496	2215	40.30%	2259	41.10%	3530	787	22.29%	833	23.60%
ixgbe	13234	4222	31.90%	4301	32.50%	7288	1414	19.40%	1479	20.29%
i40e	9666	3557	36.80%	3886	40.20%	4882	1089	22.31%	1255	25.71%
tg3	7865	2580	32.80%	2658	33.80%	4990	983	19.70%	1043	20.90%
bnx2	3856	1828	47.41%	1859	48.21%	2217	643	29.00%	687	30.99%
8139cp	856	498	58.18%	506	59.11%	314	117	37.26%	126	40.13%
r8169	2596	1241	47.80%	1264	48.69%	848	294	34.67%	319	37.62%

6.5.5 Coverage Improvement

As shown in Table 6.5, the generated fault scenarios led to decent test coverage improvement. Our approach focuses on the error handling mechanism and capability of device drivers. The error handling code only takes up a small portion of driver code. Even if we can trigger all error handling mechanisms in a driver, it does not mean that the improved coverage is very high.

As shown in Table 6.5, the improved coverage is from 0.1% to 6.5%. However, our approach can cover a lot of error handling branches. Particularly, for *iwlwifi* and *i40e*, the statement coverage can be improved by more than 200 new statements and the branch coverage can be improved by more than 150 new branches. After going through all 150 new branches, we found that most of them are error handling branches.

6.5.6 Evaluation against Other Fault Injection Techniques

We have evaluated other fault injection techniques. Their comparison with our approach can be founded in Section 6.6.3.

6.5.7 Further Potentials

Although our approach is only evaluated on Linux device drivers in our experiments, the idea can be applied in other domains. We list three potential applications in the following:

1)Linux kernel module testing. While ADFI mainly focuses on device drivers, the principles can easily apply to other kernel modules. The only effort is to identify necessary categories of target functions for different kernel modules.

2)Windows driver testing. The Windows drivers have similar structures to Linux drivers. Once we can figure out how to migrate ADFI into the Windows environment, it can be used for Windows driver robustness testing.

3) User-level program/library testing. The user-level program/library needs to invoke certain functions which can fail at runtime, for example "*malloc*" function. Our idea can be further applied to test the robustness of user-level program/library to improve reliability.

6.6 RELATED WORK

There has been much research on device driver testing since drivers account for a major portion of operating systems and are a major cause of operating system crashes [28]. Our work is related to past work in several areas, including static analysis, reliability testing and fault injection.

6.6.1 Static Analysis

Model checking, theorem proving, and program analysis have been used to analyze device drivers to find thousands of bugs [7, 26, 42, 64]. Nevertheless, these tools take time to run and the results require time and expertise to interpret. Thus, these tools are not well suited to the frequent modifications and tests that are typical of initial code development. Numerous approaches have proposed to statically infer so-called protocols, describing expected sequences of function calls [26, 42, 66]. These approaches have focused on sequences of function calls that are expected to appear within a single function, rather than the specific interaction between a driver and the rest of the kernel.

Some safety holes in drivers can be eliminated by the use of advanced type systems. For example, Bugrara and Aiken propose an analysis to differentiate between safe and unsafe userspace pointers in kernel code [15]. They focus, however, on the entire kernel, and thus may report to the driver developer about faults in code other than his own.

6.6.2 Reliability Testing

There has been much research for operating systems reliability testing [4, 14, 21, 25, 37, 68, 69, 71]. Reliability testing of operating systems has been focused on device drivers since drivers are usually developed by a third party. Previous research on device driver reliability has mainly targeted detecting, isolating, and avoiding generic programming errors and errors in the interface between the driver and the OS.

6.6.3 Fault Injection Techniques

In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed. Fault injection techniques are widely used for software and system testing [29, 54, 55, 56, 61], ranging from testing the reliability of device drivers to testing operating systems, embedded systems and real-time systems [5, 16, 35, 36, 46, 58, 67].

There are several fault injection frameworks provided on both Windows and Linux platforms.

Windows Driver Verifier: Driver Verifier provides options to fail instances of the driver's memory allocations, as might occur if the driver was running on a computer with insufficient memory. This tests the driver's ability to respond properly to low memory and other low-resource conditions.

Linux Fault Injection Framework: This framework [23] can cause memory allocation failures at two levels: in the slab allocator (where it affects *kmalloc* and most other small-object allocations) and at the page allocator level (where it affects everything, eventually). There are also hooks to cause occasional disk I/O operations to fail, which should be useful for filesystem developers. In both cases, there is a flexible runtime configuration infrastructure, based on debugfs, which will let developers focus fault injections into a specific part of the kernel.

KEDR Framework: KEDR [67] is a framework for dynamic (runtime and post mortem) analysis of Linux kernel modules, including device drivers, file system modules, etc. The components of KEDR operate on a kernel module chosen by the user. They can intercept the function calls made by the module and, based on that, detect memory leaks, simulate resource shortage in the system as well as other uncommon situations, save the information about the function calls to a kind of "trace" for future analysis by the user-space tools.

There are three major limitations in the frameworks above. First, these frameworks mainly support memory-related fault injection to simulate low resource situations. Second, these frameworks mainly provide random fault simulation. Third, these frameworks require high manual efforts. Our approach extends the above framework to support more fault situations, such as DMA-related operations and PCI-related operations. Our approach provides an easy-to-use approach with little human effort which can systematically enumerate different kinds of fault scenarios to guide fault simulation.

6.7 SUMMARY

In this chapter, we presented an approach to runtime fault injection for driver robustness testing. We have evaluated our approach on 12 widely-used device drivers. Our approach was able to generate and inject effective fault scenarios in a modest amount of time using the trace-based iterative fault generation strategy.

We have detected 28 bugs which have been further validated by manually injecting these bugs into device drivers. We have also measured test coverage and found that ADFI led to decent improvement in statement and branch coverage in drivers.

Chapter 7

CONCLUSION AND FUTURE RESEARCH

7.1 CONCLUSION

Post-silicon validation has become a critical problem in the product development cycle, driven by increasing design complexity, higher level of integration and decreasing time-to-market. According to recent industry reports, validation accounts for a large portion of overall product cost. Post-silicon validation consumes an increasing share of the overall product development time [75]. This demands innovative approaches to speeding up post-silicon validation and reducing its cost.

To accelerate post-silicon validation, this dissertation research has successfully developed several approaches and tools to shift-left post-silicon validation with virtual prototypes.

Coverage Analysis of Post-silicon Tests. Post-silicon validation tests should be well evaluated before they are issued to a silicon device. We have developed an approach to early coverage evaluation of post-silicon validation tests with virtual prototypes, which fully leverages the observability and traceability of virtual prototypes.

The approach utilizes virtual prototype coverage to estimate silicon device functional coverage. Two kinds of coverage metrics have been employed to the evaluation process. Typical software coverage metrics have been adopted to give basic coverage indication. Two hardware-specific coverage metrics, register coverage and transaction coverage, have been developed to deliver more accurate hardware-oriented coverage results.

The approach has been used for evaluating a suite of common tests on virtual prototypes of five network adapters. High confidence has been established in fidelity of coverage evaluation by further conducting coverage evaluation and conformance checking on silicon devices. With this early coverage estimation, it can guide further test generation.

Automatic Concolic Test Generation. High-quality tests should be ready before a silicon device becomes available [57] in order to save time spent on preparing, debugging and fixing tests in the post-silicon stage after the device is available. We have presented an automatic concolic approach to generation of post-silicon tests with virtual prototypes.

The approach takes advantage of white box nature of virtual prototypes and applies "concolic" idea from software testing to hardware domain. We have evaluated our approach on virtual devices for three popular network adapters. Our ACTG approach was able to generate effective test cases in a modest amount of time using the transaction-based test selection strategy. We have evaluated this strategy from two aspects: state selection efficiency and redundancy identification. The results show that our strategy performs significantly better than random selection of states under test and has significant reduction on the number of tests that have to be applied within limited amount of time.

We have measured test coverage and found that ACTG led to major improvement in coverage of device functionalities. Moreover, we applied generated test cases to both virtual and silicon devices and conducted consistency checking between their states. We have detect 20 inconsistencies between virtual and silicon devices, each of which reveals a defect in either the virtual or silicon device.

Effective Fault Injection for Driver Robustness Testing. Besides the coverage evaluation and test generation for hardware validation, we have further explored how to support software validation with virtual prototypes. Since virtual prototypes enable early driver testing, we have developed an automatic fault injection approach for driver robustness testing.

The approach can be used on both virtual platform and physical platform. We have evaluated our approach on 12 widely-used device drivers. Our approach was able to generate and inject effective fault scenarios in a modest amount of time using the iterative trace-based fault generation strategy. We have detected 28 bugs which have been further validated by manually injecting these bugs into device drivers. We have also measured test coverage and found that ADFI led to decent improvement in statement and branch coverage over error handling code in drivers.

7.2 FUTURE RESEARCH

The dissertation has presented several approaches to accelerate post-silicon functional validation with virtual prototypes. Moreover, virtual prototypes can enable more interesting research which should be explored.

7.2.1 Conformance Checking between Virtual Prototype and Hardware Design

Virtual prototype and the corresponding hardware design are developed according to the same specification. At the transaction level, they should behave the same and provide the same functionalities. However, there are some differences because they are developed separately by different software developers and hardware engineers.

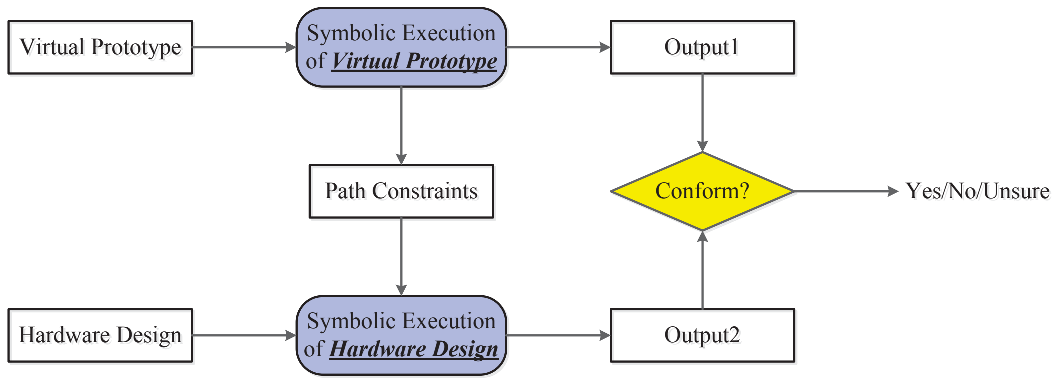


Figure 7.1: Framework for Conformance Checking between Virtual Prototype and Hardware Design

We employ symbolic execution to conduct conformance checking between a virtual prototype and the corresponding hardware design. As shown in Figure 7.1, the framework mainly includes two symbolic execution engines for virtual prototypes and hardware designs. In this dissertation, we have proposed a symbolic execution engine for virtual prototypes. Moreover, we have already developed a symbolic execution engine for hardware designs which mainly target at RTL design symbolic execution.

Using this framework, we first execute the virtual prototype with symbolic inputs using symbolic execution. For each path covered in the symbolic execution of the virtual prototype, we collect path constraints and output1. With the collected path constraints, we execute the hardware design and collect output2. Then the conformance is checked between output1 and output2.

But there are still several challenges. First, we need to figure out how to map a transaction in the virtual prototype to a sequential state transitions in the hardware design. Second, we need to solve state exploration problem in symbolic execution of virtual prototypes. Third, we need to implement a sufficient mechanism to check the conformance between output1 and output2.

7.2.2 Automatic Test Generation for RTL Simulation/Emulation with Virtual Prototypes

The automatic test generator will be extended to generate tests that can be fed into simulators and emulators. Since the designs are often simulated/emulated stand-alone, test harnesses have to be provided to feed the tests into the designs. Therefore, our test generator may have to generate such test harnesses or at least generate tests in formats that can be consumed by these harnesses.

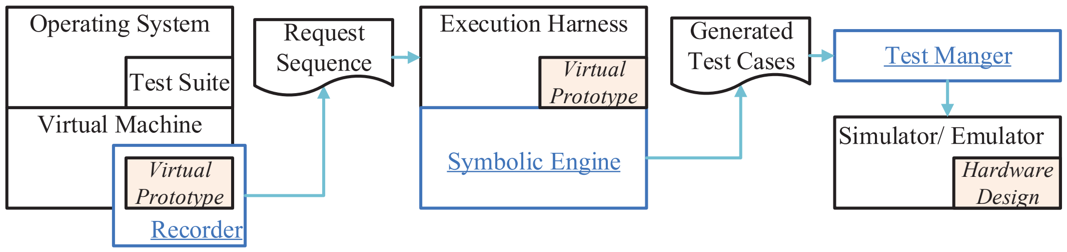


Figure 7.2: Workflow of Automatic Test Generation for RTL Simulation

As shown in Figure 7.2, our test generation algorithm will still be based on the execution of the virtual device in a virtual machine and its interaction with its software driver. Using this approach, the tests generated reflect how the software utilizes the driver. In addition, the algorithm will be guided by the coverage analysis so that both the coverage information on the virtual device and that on the hardware design will be utilized. After generating test cases, our approach can then apply a generated test case to hardware design simulation/emulation. Our automatic test generation framework includes three key components: (1) The recorder captures device requests from a concrete execution of the virtual device in the virtual machine and records them in the request sequence; (2) The symbolic engine replays the captured request sequence and generate efficient test cases based on generation strategy; (3) The test manager is an individual module which communicates with simulator/emulator to apply a test case to a hardware design.

7.2.3 Further Research on Device Fault Injection with Virtual Prototypes for Driver Testing

To ensure the system reliability, device drivers must tolerate hardware-related faults, such as DMA failures, interrupt loss, device I/O errors. Therefore, it is necessary that different kinds of hardware faults can be generated and injected to test the driver robustness.

As illustrated in Section 5.5, we have developed a framework for device fault injection with virtual prototypes. The framework first generates numerous fault scenarios automatically based on the runtime trace from concrete execution of device drivers and virtual prototypes. Generated fault scenarios can be used to simulate runtime hardware faults to test whether device drivers can handle unexpected hardware faults correctly.

Currently there are only two fault models: I/O error model and Interrupt loss model. However, there are more possible fault models like DMA error model and Interrupt hang model. In the future, we need to add these fault models into our fault injection framework.

We have conducted preliminary evaluation on three virtual prototypes and the corresponding Linux drivers using our VPFI framework. We found 2 serious bugs with generated fault scenarios. In the future work, we will continue implementing VPFI framework and applying it to more virtual prototypes and the corresponding drivers. We will work on fully developing this approach and conducting further evaluation.

REFERENCES

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[2] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann. A unified methodology for pre-silicon verification and post-silicon validation. In *DATE*, 2011.

[3] Allon Adir, Amir Nahir, Avi Ziv, Charles Meissner, and John Schumann. Reaching coverage closure in post-silicon validation. In *HVC*, 2010.

[4] A. Albinet, J. Arlat, and J-C Fabre. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *International Conference on Dependable Systems and Networks*, 2004.

[5] Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *International Conference on Dependable Systems and Networks*, 2004.

[6] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[7] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah

Ustuner. Thorough static analysis of device drivers. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[8] K. Balston, M. Karimibiuki, A.J. Hu, A. Ivanov, and S. J E Wilton. Post-silicon code coverage for multiprocessor system-on-chip designs. *IEEE Transactions on Computers*, 2011.

[9] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Structural coverage of feasible code. In *AST*, 2010.

[10] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATEC*, 2005.

[11] Darrell Bethea, Robert A. Cochran, and Michael K. Reiter. Server-side verification of client behavior in online games. *ACM Trans. Inf. Syst. Secur.*, 14(4), 2008.

[12] T. Bojan, M.A. Arreola, E. Shlomo, and T. Shachar. Functional coverage measurements and results in post-silicon validation of Core^{TM}2 duo family. In *HLVDT*, 2007.

[13] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: attacking path explosion in constraint-based test generation. In *TACAS*, 2008.

[14] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *Conference on USENIX Annual Technical Conference*, 2010.

[15] Suhabe Bugrara and Alex Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy*, 2008.

[16] Gianpiero Cabodi, Marco Murciano, and Massimo Violante. Boosting software fault injection for dependability analysis of real-time embedded applications. *ACM Trans. Embed. Comput. Syst.*, 2011.

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[18] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. Coverage evaluation of post-silicon validation tests with virtual prototypes. In *DATE*, 2014.

[19] Kai Cong, Fei Xie, and Li Lei. Automatic concolic test generation with virtual prototypes for post-silicon validation. In *ICCAD*, 2013.

[20] Kai Cong, Fei Xie, and Li Lei. Symbolic execution of virtual devices. In *QSIC*, 2013.

[21] D. Cotroneo, D. Di Leo, F. Fucci, and R. Natella. Sabrine: State-based robustness testing of operating systems. In *International Conference on Automated Software Engineering*, 2013.

[22] Lin David, Hong Ted, Fallah Farzan, Hakim Nagib, and Mitra Subhasish. Quick detection of difficult bugs for effective post-silicon validation. In *DAC*, 2012.

[23] Roberto Jung Drebes and Takashi Nanya. Limitations of the linux fault injection framework to test direct memory access address errors. In *IEEE Pacific Rim International Symposium on Dependable Computing*, 2008.

[24] R. Drechsler, S. Eggersgluss, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille. On acceleration of SAT-Based ATPG for industrial designs. *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008.

[25] Joao Duraes and H. Madeira. Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. *Transactions of IEICE*, 2003.

[26] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM Symposium on Operating Systems Principles*, 2001.

[27] Bellard Fabrice. QEMU. http://wiki.qemu.org/Main_Page, 2013.

[28] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows xp kernel crash analysis. In *Conference on Large Installation System Administration*, 2006.

[29] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, 2013.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, 2005.

[31] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *ACM Queue - Networks*, 2012.

[32] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[33] T. Hong, Y Li, Sung-Boem Park, D. Mui, D. Lin, Z.A. Kaleq, N. Hakim,

H. Naeimi, D.S. Gardner, and S Mitra. QED: quick error detection tests for effective post-silicon validation. In *Test Conference (ITC)*, 2010.

[34] International Business Strategies, Inc. Global systemIC industry service monthly reports. http://www.ibs-inc.net, 2014.

[35] Tahar Jarboui, Jean Arlat, Yves Crouzet, and Karama Kanoun. Experimental analysis of the errors induced into linux by three fault injection techniques. In *International Conference on Dependable Systems and Networks*, 2002.

[36] Andras Johansson and Neeraj Suri. On the impact of injection triggers for os robustness evaluation. In *International Symposium on Software Reliability Engineering*, 2007.

[37] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[38] J. Keshava, N. Hakim, and C. Prudvi. Post-silicon validation challenges: How EDA and academia can help. In *DAC*, 2010.

[39] James C. King. Symbolic execution and program testing. *Commun. ACM*, 1976.

[40] H.F. Ko and N. Nicolici. Automated trace signals identification and state restoration for improving observability in post-silicon validation. In *DATE*, 2008.

[41] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *IEEE Asian Test Symposium*, 2010.

[42] J.L. Lawall, J. Brunel, N. Palix, R.R. Hansen, H. Stuart, and G. Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *International Conference on Dependable Systems Networks*, 2009.

[43] Li Lei, Kai Cong, and Fei Xie. Optimizing post-silicon conformance checking. In *ICCD*, 2013.

[44] Li Lei, Kai Cong, Zhenkun Yang, and Fei Xie. Validating direct memory access interfaces with conformance checking. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, 2014.

[45] Li Lei, Fei Xie, and Kai Cong. Post-silicon conformance checking with virtual prototypes. In *DAC*, 2013.

[46] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Conference on Design, Automation and Test in Europe*, 2009.

[47] Jin-Fu Li, Yung-Fa Chou, Chih-Yen Lo, Che-Wei Chou, Ding-Ming Kwai, Yun-Chao Yu, and Cheng-Wen Wu. A built-in self-test scheme for 3d rams. In *International Test Conference (ITC)*, 2012.

[48] Xijiang Lin, Kun-Han Tsai, Chen Wang, M. Kassab, J. Rajski, T. Kobayashi, R. Klingenberg, Y. Sato, S. Hamada, and T. Aikyo. Timing-aware ATPG for high quality at-speed testing of small delay defects. In *Asian Test Symposium*, 2006.

[49] Linux. Fault Injection Capabilities Infrastructure. http://lxr.linux.no/linux+v3.14/Documentation/fault-injection/.

[50] Xiao Liu and Qiang Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *DATE*, 2009.

[51] Xiao Liu and Qiang Xu. On signal selection for visibility enhancement in trace-based post-silicon validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[52] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, 2011.

[53] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[54] Paul D. Marinescu, Radu Banabic, and George Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Conference on USENIX Annual Technical Conference*, 2010.

[55] Paul D. Marinescu and George Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 2011.

[56] Paul Dan Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *Dependable Systems and Networks*, 2009.

[57] S Mitra, S.A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *DAC*, 2010.

[58] Abbas Mohammadi, Mojtaba Ebrahimi, Alireza Ejlali, and Seyed Ghassem Miremadi. Scfit: A FPGA-based fault injection technique for SEU fault model. In *Conference on Design, Automation and Test in Europe*, 2012.

[59] A. Nahir, A. Ziv, M. Abramovici, A. Camilleri, R. Galivanche, B. Bentley, H. Foster, A. Hu, V. Bertacco, and S. Kapoor. Bridging pre-silicon verification and post-silicon validation. In *DAC*, 2010.

[60] A. Nahir, A. Ziv, M. Abramovici, A. Camilleri, R. Galivanche, B. Bentley, H. Foster, A. Hu, V. Bertacco, and S. Kapoor. Bridging pre-silicon verification and post-silicon validation. In *DAC*, 2010.

[61] Thomas Naughton, Wesley Bland, Geoffroy Vallee, Christian Engelmann, and Stephen L. Scott. Fault injection framework for system resilience evaluation: Fake faults for finding future failures. In *Workshop on Resiliency in High Performance*, 2009.

[62] Shannon Nelson and Peter Waskiewicz. Virtualization: Writing (and testing) device drivers without hardware. In *Linux Plumbers Conference*, 2011.

[63] Sung-Boem Park, T. Hong, and S Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *IEEE Transactions on CADICS*, 2009.

[64] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for linux device driver verification. In *International Conference on Integrated Formal Methods*, 2007.

[65] Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *International Symposium on Empirical Software Engineering and Measurement*, 2011.

[66] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *International Conference on Software Engineering*, 2007.

[67] Vladimir V. Rubanov and Eugene A. Shatokhin. Runtime verification of linux kernel modules based on call interception. In *International Conference on Software Testing, Verification, and Validation*, 2011.

[68] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *ACM European Conference on Computer Systems*, 2009.

[69] Leonid Ryzhyk, John Keys, Balachandra Mirla, Arun Raghunath, Mona Vij, and Gernot Heiser. Improved device driver reliability through hardware verification reuse. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[70] Pradyumna Sampath and Bangalore Rachana Rao. Efficient embedded software development using QEMU. In *13th Real Time Linux Workshop*, 2011.

[71] C. Sarbu, A. Johansson, N. Suri, and N. Nagappan. Profiling the operational behavior of os device drivers. In *International Symposium on Software Reliability Engineering*, 2008.

[72] Constantin Sârbu, Andréas Johansson, Falk Fraikin, and Neeraj Suri. Improving robustness testing of cots os extensions. In *Proceedings of the Third International Conference on Service Availability*, 2006.

[73] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *ESEC/FSE*, 2005.

[74] Varshapriya Shakti D Shekar, B B Meshram. Device driver fault simulation using kedr. *International Journal of Advanced Research in Computer Engineering and Technology*, 2012.

[75] E. Singerman, Y. Abarbanel, and S. Baartmans. Transaction based pre-to-post silicon validation. In *DAC*, 2011.

[76] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *International Symposium on Software Testing and Analysis*, 2010.

[77] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 2006.

[78] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 2004.

[79] I. Wagner and V. Bertacco. Reversi: Post-silicon validation system for modern microprocessors. In *ICCD*, 2008.

[80] Seongmoon Wang. A bist tpg for low power dissipation and high fault coverage. *IEEE Trans. Very Large Scale Integr. Syst.*, 2007.

[81] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *International Symposium on Software Testing and Analysis*, 2008.

[82] Wikipedia. Stack trace. http://en.wikipedia.org/wiki/Stack_trace.

[83] Windows. Low Resources Simulation. http://msdn.microsoft.com/en-us/library/windows/hardware/ff548288(v=vs.85).aspx.