UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
Dottorato di Ricerca in Ingegneria Informatica ed Automatica

A. D. MCCXXIV

Comunità Europea
Fondo Sociale Europeo

MODEL-BASED VERIFICATION OF OPERATING SYSTEMS DEVICE DRIVERS

FRANCESCO FUCCI

Tesi di Dottorato di Ricerca
(XXVII Ciclo)
Marzo 2015

Il Tutore	Il Coordinatore del Dottorato
Prof. Domenico Cotroneo	Prof. Francesco Garofalo

Dipartimento di Ingegneria Elettrica
e delle Tecnologie dell'Informazione

🖃 Via Claudio, 21 - 80125 Napoli - ☎ *[+39]* 081 76 83813 - 🗐 *[+39]* 081 76 83816

MODEL-BASED VERIFICATION OF OPERATING SYSTEMS DEVICE DRIVERS

By
Francesco Fucci

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
VIA CLAUDIO 21, 80125, NAPLES, ITALY
MARCH 2015

‘‘To my mother, for her constant support.”

Abstract

Critical systems depend on software more than ever. In particular, off-the-shelf operating systems (OS) have a central role in the development of critical systems. Unfortunately, OS are plagued by software defects that threaten their reliability, since verification techniques are still not enough cost-efficient to prevent such defects. In particular, empirical studies found that defective device drivers are the major cause of failures of operating systems. Therefore, more sophisticated techniques are needed in order to make the verification of device drivers more cost-efficient.

This thesis addresses this problem by proposing three solutions for detecting software defects in device drivers. The thesis first proposes a methodology that enhances symbolic execution with model-based verification. The idea is that the developer provides a model of expected interactions between the device driver, the operating system and the device, based on documentation and domain expertise about the OS and the device. We propose a language (*SLANG*) and a run-time support (*SymCheck*) to ease the developer in specifying behavioral models and checking them through symbolic execution.

The second contribution of this thesis is an enhanced platform for improving the speed of symbolic execution. This platform is based on an efficient representation of intermediate code, which can achieve an average speed-up of 3x compared to a state-of-the-art symbolic execution platform.

Finally, this thesis provides an approach for run-time verification of the behavior of device driver. The idea is that device drivers behave correctly in most cases, and that anomalies in their behavior are symptomatic of failures. Therefore, the approach uses failure-free execution traces of the device driver, to extract a model of the expected behavior. Then, this model is used to generate a device driver monitor, which inspects the state of the device driver in order to detect divergences between the expected behavior and the actual one.

The proposed approaches have been applied on device drivers from the Windows and Linux OS, showing their applicability and usefulness on real-world off-the-shelf OS.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Molte persone mi hanno sopportato e “supportato” nel corso di questi tre anni. Sembrerá superfluo dire che ho imparato molto in questi anni, non solo dal punto di vista scientifico ma anche dal punto di vista umano. Forse un paio di pagine non sono sufficienti per ringraziare tutte le persone che mi hanno aiutato a crescere e maturare sia dal lato lavorativo e accademico sia lato di vita personale. In primis, ringrazio il mio advisor Domenico che mi ha lasciato tutta la libertá che volevo, forse anche piú di quella che meritavo. Un ulteriore ringraziamento va al prof. George Candea che mi ha ospitato nel suo laboratorio a Losanna e mi ha reso partecipe di moltissime attivitá interessanti che ho trovato estremamente formative. Un particolare grazie va a Roberto che mi ha seguito con estrema pazienza, in particolare negli ultimi mesi di stesura della tesi. Grazie va anche ad Antonio che mi ha dato molti consigli e ho scambiato molte opinioni nel corso di questi anni. Ringrazio tutti i ragazzi del gruppo di ricerca che sono stati ottimi compagni di avventura alcuni che conosco dall’inizio altri che si sono uniti alla squadra nel corso di questi anni: Anna, Flavio, Domenico, Roberto, Fabio, Raffaele, Ugo, Luigi, Alma, Luca, Ken, Salvatore, Mario, Daria, Emanuela. Spero di non dimenticare nessuno :)

Nella mio soggiorno in Svizzera ci sono state alcune persone fondamentali nella mia vita di tutti i giorni. In primis, le mie cugine Maria Carmela,Consuelo e Caterina con Fabio ed Enzo che con me sono stati fantastici. Poi ringrazio di cuore Matteo che é stato un ottimo amico e compagno. Un altro sentito grazie va ai ragazzi del DSLAB dell’EPFL Cristian, Silviu, Loic, Amer, Stefan, Baris, Jonas e Vova che mi hanno ospitato e coinvolto sin da subito, ricordo con piacere le tante sfide a FIFA 2014 e cene in giro per Losanna. Ringrazio in particolare Vitaly per avermi aiutato e insegnato molte cose per tutto il periodo della mia internship.

Dal lato personale, devo ringraziare altrettante persone. Inizio con la mia famiglia che mi ha aiutato a superare i momenti piú bui che ho attraversato soprattutto nell'ultimo anno e mezzo. Mia madre che mi ha sempre ascoltato ed é stata sempre la prima a credere in ogni mia iniziativa, mio padre che ha cercato sempre di farmi ragionare data la mia estrema impulsivitá. I miei fratelli Luca, Cristina e Valentina che sono le risorse piú preziose che ho, che mi sono sempre accanto. I miei amici di sempre Mario, Delio e Salvatore che insieme a mio fratello hanno sopportato negli ultimi mesi la mia assenza e irritabilitá. Nonostante tutto un grazie va anche a Claudia che mi ha ascoltato anche quando c'era tutt'altro nella sua testa. Infine voglio ringraziare anche Alessia che negli ultimi mesi mi é stata molto vicina.

Con affetto,
Francesco

Chapter 1

Introduction

1.1 Motivation

Modern critical systems differ considerably compared to critical systems developed twenty-years ago. Nowadays, critical systems offer plenty of more sophisticated functionalities. Smartgrids, which allow an intelligent and autonomous management of electrical energy dispatching, are replacing old electrical infrastructure. Autonomous driving is becoming a real opportunity and it is emerging progressively. Google estimated that in 2017-2020 self-driving car are going to be available on the market [1]. The implementation of such sophisticated functionalities require more complex code that continuously grows in terms of program size and interactions across components.

The augment of software complexity entails the increase of software defects. It seems that the relationship between the program size and the number of software defects is roughly linear, even in mature software [2]. Further studies reported the average number of software defects in real code. In code used in business critical systems, the average number of errors is about 15-50 errors per 1000 lines of delivered code. Microsoft reported about 10-20 defects per KLOC during in-house testing, and 0.5 defect per KLOC in released product [3]. A report shows that in spacecraft software the number of defects is 3 per KLOC during in-house testing and 0.1 defect per KLOC in released products [4]. NASA reported an average of 0.0004 defects per KLOC.

Despite the advances in verification methodologies and program verification tools, even in safety critical systems, software defects still remain one of the main cause of computer systems failures. For this reason, safety critical standards impose strict requirements about software dependability, demanding very expensive verification activities to satisfy certification criteria. The standards ISO26262 [5] and DO-178B [6] define strict verification requirements. The level C of DO-178B certification involves both coverage of low-level requirements and exhaustive structural testing, that is, achieving 100% statement source code coverage. For higher levels of certification the verification requirements are even more strict. System testing is the most expensive development activity. Normally, this is about 40% of the total development costs but for some critical systems it is likely to be at least 50% of the system development costs [7].

To reduce the costs of reimplementing functionalities from scratch, the current trend is to build safety critical system using OTS (off-the-shelf) components. In particular,the *operating system* (OS) is one of the main OTS component used everywhere in all modern software systems. Unfortunately, requirements dictated by the standards clash with the huge complexity of modern OS. On the one hand, the application of formal methods to prove software correctness is not practicable for the current needs: proving the correctness of seL4 OS required around 20 person years for 9,300 lines of code [8]. Hence, it is unrealistic to scale such techniques to real world software, that is hundred times larger than seL4. Today, the challenge is to devise more **cost-efficient** approaches to scale verification of OS. A study reports that the test effort increases exponentially with test coverage, i.e, the cost required to reach the 91% from the 90% of statement coverage is significantly higher with respect to the required to reach the 51% from 50% of statement coverage [9].

Several studies analyzed the **cause of failures in OS**. An experimental study performed on Windows XP shows that kernel extensions were responsible of the 85% of the entire fraction of known crashes [10]. Kernel extensions are pluggable modules that enhance OS capabilities, e.g., filesystems and device drivers. A further study reports that device drivers cause more failures than faulty hardware [11]. Device drivers are indeed the Achilles' heel of operating systems reliability. An empirical study on the Linux kernel [12] estimated that the number of defects in device driver code is 3-7 higher with respect to the other parts of the kernel. An example that shows the impact of OS failures in critical systems involved the famous Mars Pathfinder mission [13]. In the first week after the landing, the Pathfinder onboard computer, that ran VxWorks, spontaneously reset itself about half dozen times.

From that moment, every time the Pathfinder reset itself, data was delayed by a day. After three weeks of work, the team at NASA discovered that the failure was due to a *priority inversion* phenomenon [14]. According to the report, the problem did not manifest in the verification phase because the antenna, that sent data on Earth, behaved faster than was expected in the design phase. The RTOS (Real-Time Operating System) was unable to give the right priority to the thread that sent data on Earth.

Device drivers realize the layer interposed between devices and OS kernel. The OS kernel communicates with devices through device drivers on the behalf of applications. To abstract the functionalities of classes of devices, OS kernels define interfaces. Device drivers use these interfaces to completely mask the implementation details. Thus, applications can communicate with class of devices without knowing the details regarding how the device internally works. Drivers code interacts with several kernel components such as memory management, bus and DMA interface. From empirical studies, potential defects include :

- incorrect use of OS resources such as dynamic allocated memory areas, DMA memory, spinlocks, work queues [15, 16, 17].
- mastering complex kernel API that are often not well documented [17, 18].

What makes device driver verification really complex is the dense interaction across drivers, devices and kernel. Device drivers abstract the behavior of a family of devices. A study in the Linux kernel shows that a significant portion of device drivers support dozens of devices [19]. Device vendors offer interfaces that allows to program their products according to a certain standard such as IEEE 1394, SATA, SCSI. Concerning the communication between drivers and devices, we can imagine the device interface as a union of three types of registers: (i) *state registers* , (ii) *command registers* and (iii) *data registers.*

Device drivers send commands to the devices to perform an operation. Commands, as side effect, can modify the device state. Device drivers use data registers to send data on the behalf of the OS kernel to the device. The operations that the device driver can issue *depends on the state* of the device. Therefore, given two equal inputs the device driver can can react differently depending on the state reached by the device.

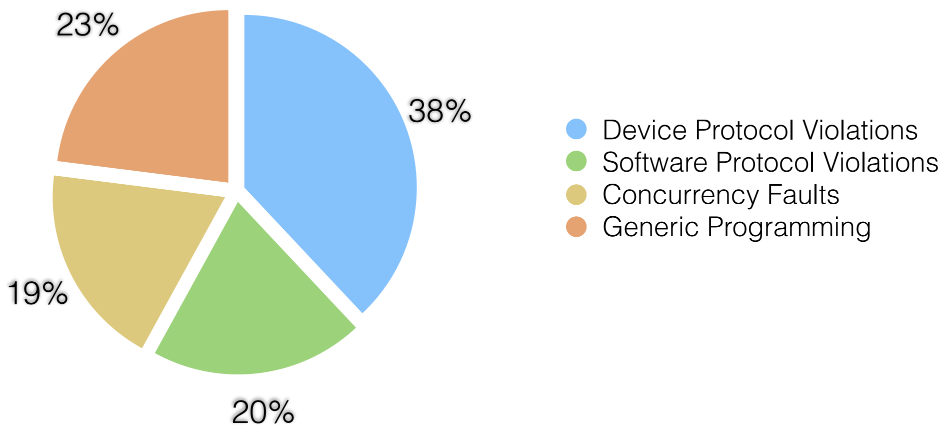


Figure 1.1: Device driver defects taxonomy in the Linux kernel.

A communication between device driver, hardware and OS must follow rules that establish the correct timing, order and format of interactions. In particular, the *device protocol* regulates communication with the hardware device, and the software protocol regulates the communication with the OS. Each device embeds a unique protocol defined by the manufacturer. OS provides a common interface to overcome the diversity of devices e.g, different Ethernet vendors. Figure 1.1 shows a taxonomy of device drivers defects in the Linux kernel [17]. The study was performed on a complete kernel development history for 13 device drivers acting on PCI, IEEE 1394, and USB bus types. The total defect database consisted of 498 defects. Device driver defects were classified in four categories:

- *Device protocol violations:* occur when the device driver violates the hardware protocol, and may lead in a failure of the hardware to provide its required service. Examples of such kind of defects are are such that put the device in an incorrect state, mis-interpreting the device state, issuing a sequence of commands to the device that violates the device protocol, specifying incorrect timeout values. Device protocol violations are the 38% of the defects stored in the database.

- *Software protocol violations:* occur when the device driver violates the required protocol with the OS. All violations of expected ordering,format or timing in interactions between the OS and the device driver. Ordering violations ,for instance, include forgetting to wait for a completion callback, forgetting to release a resource or releasing a resource in the wrong order, and falsely returning a success status from an operation that failed. Software protocol violations are 20% of the defects stored in the database. An example of OS protocol is one required to use DMA addresses, which are a limited resource of the OS [20]. When a device driver maps a DMA address space, it must eventually unmap it, in order to avoid the waste of DMA addresses.

- *Concurrency defects:* occur when a device driver incorrectly synchronizes multiple threads, causing a race condition or deadlock. These kind of defects are not related to a particular aspect of the driver functionality, but rather to the model of computation enforced by the OS on device drivers. All the device drivers are involved in concurrent activities such as handling I/O requests, processing interrupts, and dealing with power management and hot-plugging activities. Concurrency defects are the 19% of the defects stored in the database.

- *Generic programming defects:* this category includes common coding errors, such as memory allocation errors, typos, missing return value checks, and program logic errors. Generic programming defects are the 23% of the defects stored in the database.

Concurrency is a further source of complexity. Typically a device driver stands in an extremely concurrent environment. Subtle synchronization issues may lead to bugs such as race conditions and deadlocks. These bugs appear in particular kernel thread interleavings. Furthermore, device drivers are the major contributor of the entire OS kernel source code. Latest version of the Linux kernel (3.19-rc5 reported at the time of this thesis) has over 10 million LOC code whereas the number of drivers LOC is around 6.6 million. Approximately the 66% of the Linux kernel code is device drivers. All these aspects contribute in making the development and the testing of device driver a stressful endeavor.

1.2 State of the Art: Verification of Device Drivers

Assessing device drivers is a stressful and error-prone task. Researchers developed several approaches to augment the effectiveness of device drivers verification techniques. Verification checks that a product i.e., OS component satisfies a set of design requirements, e.g., the driver sends correctly a write command. Today, developers start using verification techniques that mix static/dynamic analysis techniques to find in device drivers code bugs causing *memory leaks*, *null pointer dereferences*, *race conditions* and *deadlocks*. In the following section we describe the main approaches for assessing device drivers.

1.2.1 Device Drivers Testing

Device drivers accept inputs from two sources: (i) the OS kernel, (ii) the device. Figure 1.2 depicts how the device driver layer interacts with the OS kernel and the hardware layer. Devices store commands and states inside their internal registers and trigger signals that are accepted by the OS kernel and forwarded to the device driver. Conversely, OS kernel invokes device drivers entry points to satisfy application requests. Thus, to test a device driver, a developer should provide a set of inputs that come both from the device and the OS kernel.

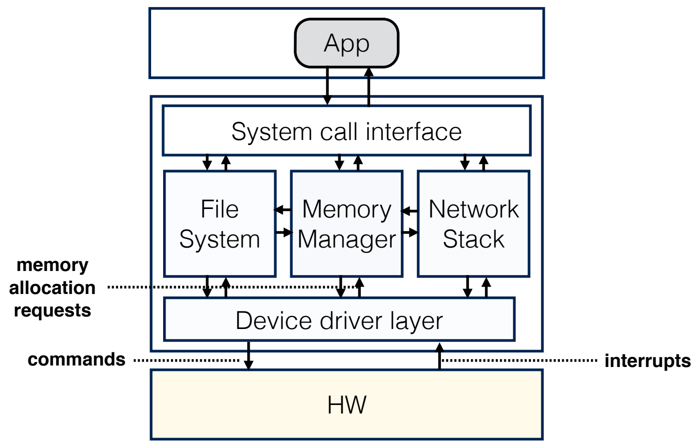


Figure 1.2: Harness for device driver testing.

Generally speaking, testing is not exhaustive and it is unsatisfying for several reasons. The current state of the practice is to test the device driver behavior under high system loads. The Linux Test Project [21], for instance, aims at delivering test suites to the open source community that validate the reliability, robustness, and stability of Linux. LTP is a joint project started by SGI, developed and maintained by IBM, Cisco, Fujitsu, SUSE, Red Hat. Testsuite contains a collection of tools for testing the Linux kernel and related features, e.g, disk stress test.

Other approaches, based on fault injection, simulate device faulty behavior. D-Cloud [22] implements FaultVM that emulates faulty hardware. The fault injector of FaultVM is implemented in the same layer of the virtual hardware. When a faulty instruction is executed the system can inject faults in two ways: (i) placing I/O hooks on read/write operations that modify the read/written value to a faulty one; (ii) the direct modification within the virtual hardware device.

Testing techniques require a tremendous effort. The first factor is that use ineffectively the resources because the test harness randomly chooses inputs and events. Thus, hours of testing may lead in covering always the same code portions.

1.2.2 Static Analysis

The idea behind static analysis is to analyze all the paths of a program without actually executing the code. By providing specifications about the behavior of a program, developers achieve report with summaries about **potential** problems inside the code. Key approaches, in the static analysis of device drivers area, are three: Metal [23], SDV [18], Carburizer [24].

Metal introduces *meta-level compilation*, a language that allows to specify rules at compiler level. The MC compiler extension checks the code and it verifies that the code obeys to the declared rule. A prototype of MC was an extension of GCC with the capability of extra functionalities that enable a custom plugin to attach to the abstract syntax tree and verify protocols.

SDV engine checks *temporal safety* properties of sequential C programs. SDV uses SLAM a static analysis engine to find kernel API usage errors. A temporal safety property is violated iff. there exists a finite execution path in which the property is not valid. The API usage rule is described through a model that has a static set of state variables and a set of event and state transitions on event. To verify the device driver, SDV uses the

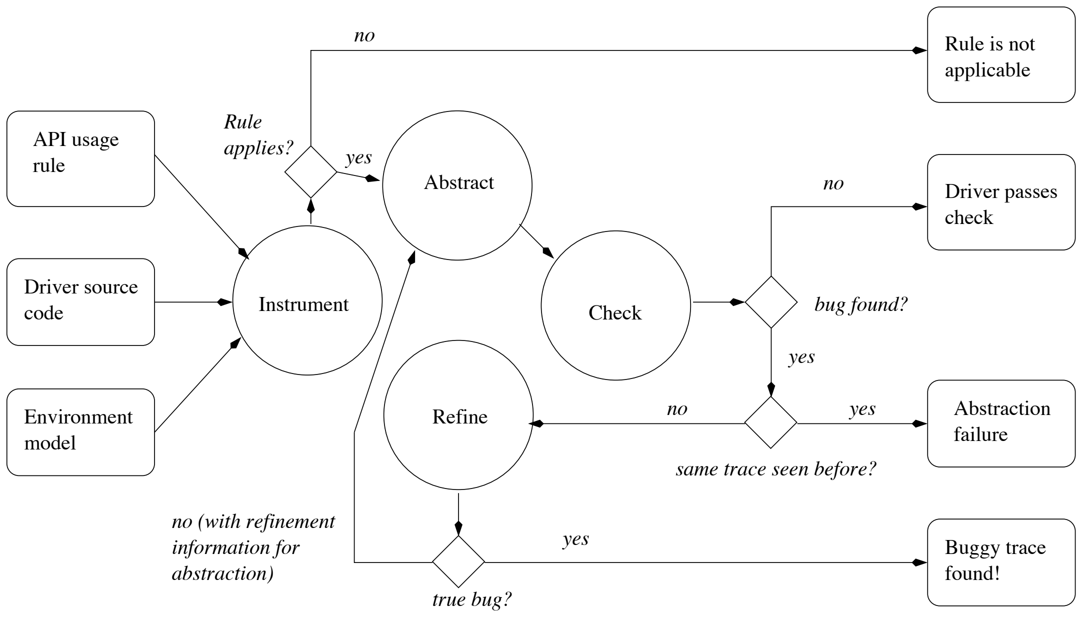


Figure 1.3: SDV approach.

counterexample-guided abstraction refinement that automatically infers an abstract model of the original program. This model has a sufficient degree of precision for proving the program correctness with respect to an API usage rule.

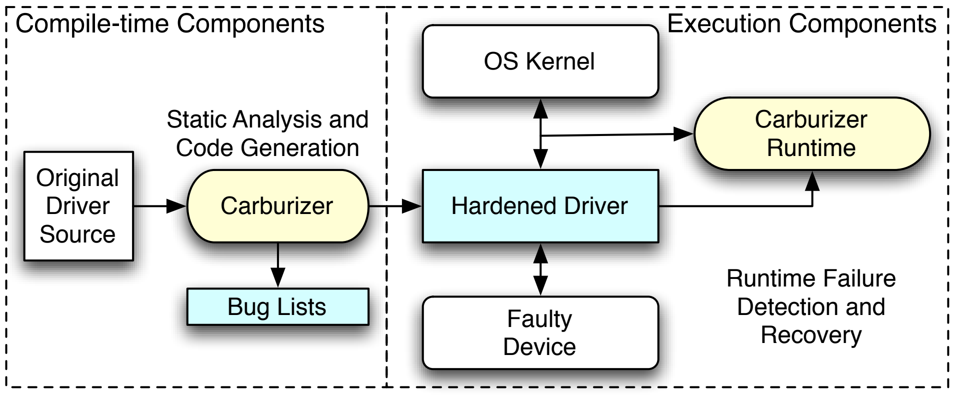


Figure 1.4: Carburizer approach.

Carburizer [24] is a code manipulation tool with an associated runtime support that modifies drivers to survive hardware failures. Figure 1.4 depicts the Carburizer approach for hardening device driver code to tolerate device fault. It starts by analyzing driver code to find where the device driver accepts input from the external word. The tool makes sure that the input device data has checks for correctness, otherwise it rewrites the code to insert input validation checks. Then, if the data is checked for correctness Carburizer inserts code that returns an error if the data is incorrect. Runtime support detects stuck interrupts and non-responsive devices.

The main drawback of static analysis approaches is the high number of false positives and false negatives. Since static analysis requires a model of the environment e.g., memory, I/O, OS API, the precision of the analysis depends on the precision of the model. Creating a precise model is a complex and error-prone task. Today, those techniques are not widely used because they require a lot of manual inspection to filter out false positives and false negatives.

1.2.3 Symbolic Execution

A program provides outputs in relation with the input that receives from the external world. Each input exercises a specific sequence of instructions, that executed, bring the program state in a certain point of the set of all possible program states.

In a non-concurrent environment the instruction flow is sequential except into specific decision points in which the execution changes according the input, namely *branch points.*

```
typedef enum States { S1, S2 , UNKNOWN };
static States var_state;
var_state = UNKNOWN;

int computeValue(int x){
    if(x > 5){
        var_state = S1;
        return x − 1;
    }else if(x < 1){
        var_state = S2;
        return x + 1;
    }else{
        var_state = UNKNOWN;
        return 0;
    }
}
```

Figure 1.5: A sample program for introducing symbolic execution tree.

Figure 3.10 depicts a sample program that has three execution paths. This function is composed by two if statement in series, therefore the number of execution paths is three. The first path is exercised when the value $x \in [6, +\infty[$ whereas the second path is exercised when $x \in]-\infty, 0[$ and the third when $x \in [1, 5]$. On the first path the value of internal variable $var\_state = S1$, whereas on the second path $var\_state = S2$ and on third remains $var\_state = UNKNOWN$. The number of execution paths grows exponentially with the number of the program branch points.

Symbolic execution performs an enumeration of all the feasible paths in a program [25] . A symbolic execution engine creates a *symbolic execution tree* that contains all the paths that the program can execute. To generate the symbolic execution tree, the symbolic execution engines treats input variables as union of all possible concrete values that x can assume ϕ, that we denote with, this process is known as marking the variable as symbolic.

In figure 1.5 the variable x marked as symbolic, becomes the symbolic variable ϕ that represent all the possible finite precision integers e.g., 32-bit and 64-bit. When the symbolic execution engine reaches a branch point and the outcome of the decision depends on the symbolic input , it *forks*. At this point we can ideally proceed along two paths, the first exercised when the constraint is true and the second when the constraint is false. Forking consists in making a copy of the entire program state i.e., constraints, program counter, registers, stack, etc into a new program state. The two states differ for a constraint. Thus, after forking, if the constraint solver replies with the feasibility of the constraints, the symbolic execution engine maintains two copies of the program state. At this point the execution can continue along both paths. A scheduling policy determines the order in which those paths are going to be executed. This process is repeated recursively through all the program execution.

Figure 1.6 shows the symbolic execution tree for the program in figure 1.5. Each node $n \in N$ in the tree represent a program state, given N the set of all feasible states in a program. A state m follows a state n, $n \to m$, if the constraint in n is feasible. Leafs in the symbolic execution tree are the points in which the program exits. The value of the variable var_state changes according to which path is executed i.e. on each path we may have different registers and memory location values.

Constraints collected during the execution are sent to a constraint solver to decide if a

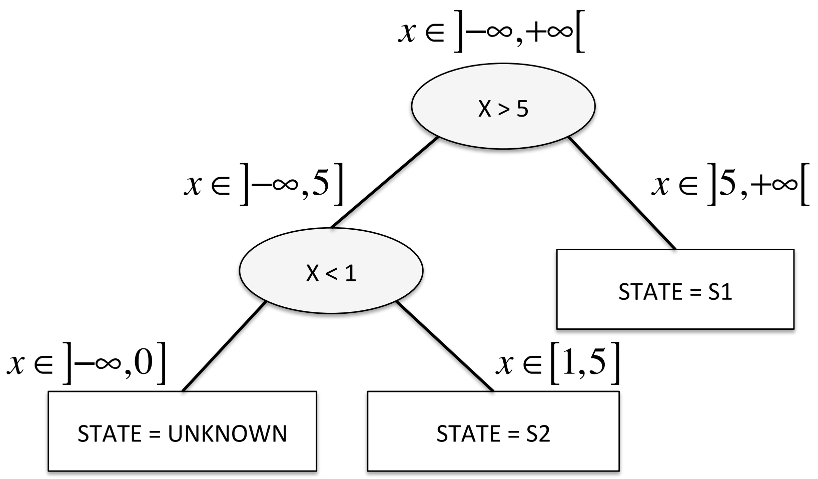


Figure 1.6: Symbolic execution tree of the example in figure 1.5.

branch is feasible or not. In figure 1.6 when the execution reaches the first branch point, the symbolic execution engines queries the constraint solver creating two constraints with the branch condition and its negated form and its negated form. In this case, since both paths are feasible the constraint solver replies saying that the constraints $x \leq 5$ and $x > 5$ are both feasible. When the engine reaches the second branch, it sends two queries to the constraint solver the first is $x \leq 5 \wedge x < 1$ and the second that is $x \leq 5 \wedge x \geq 1$. The constraints are feasible also in this case, thus the engine can proceed in the exploration of both paths. Finally, when the symbolic execution engine reaches an exit point such as a return instruction,an assert; the solver can compute the concrete values that exercise the terminated path.

Recently, symbolic execution gained a lot of attraction in the research community because it has proved to be effective in finding serious errors in programs [26, 27, 28, 29]. A key project in this area is S2E [30], a platform that performs multi-path in-vivo analysis of binary executables. *Selective symbolic execution* was born from the idea that not all the paths are interesting for an analysis. In the case of unit-testing, we are interested only in exercising the paths of the component that is subject of our analysis. Thus, we do not want to focus in exercising paths that are outside our unit. With *consistency models* S2E can narrow the set of paths on a specific subset of interest for the analysis.

Several projects are based on S2E ([29, 31, 16, 15, 32, 33]), making it the *de facto* standard for symbolic execution tools. In particular, two projects had a significant impact in the context of device driver verification. DDT [16] found several bugs such as memory leaks, race conditions on Windows closed source device drivers. DDT introduces the concept of *symbolic hardware*, to enable testing the interaction of a device with the device driver along multiple paths. In other words, this technique allows to treat devices as symbolic variable source. Conversely, Symdrive [15] uses S2E to find bugs on several Linux device drivers.

The design of S2E is depicted in figure 1.7. S2E combines virtualization, dynamic binary translation and symbolic execution to perform symbolic execution on real systems software stack, e.g., Windows/Linux based. To reach this goal, S2E combines QEMU dynamic binary translation with symbolic execution performed by KLEE [26]. Guest code is translated into the QEMU intermediate representation, namely TCG (tiny code generator). Code is partitioned in pieces named *translation blocks.* If a translation block accesses only concrete

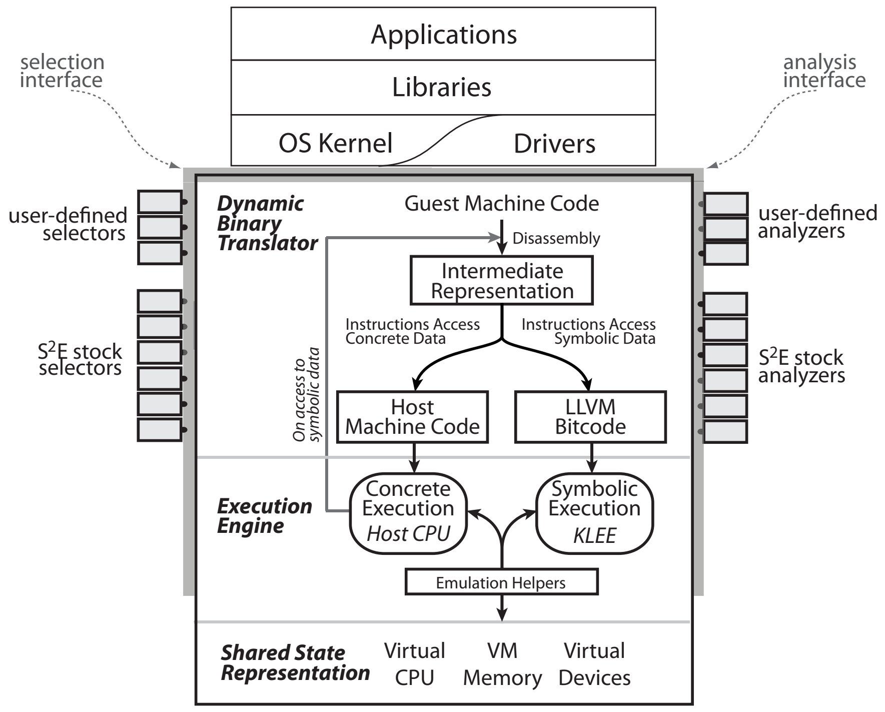


Figure 1.7: The design of the S2E platform.

data the guest code is translated into host cpu binary code and then executed as it is on the *host* CPU. Conversely, if there are accesses on symbolic data the translation block is translated into LLVM bitcode and then symbolically interpreted by KLEE. To keep the virtual machine state consistent, S2E forces synchronization QEMU and KLEE copies of the registers that are stored in different memory areas. Also the virtual memory is shared between QEMU and KLEE to allow the reading and the writing of symbolic variables. Furthermore, S2E offers a plugin infrastructures that allows to make the system extensible with the following kind of functionalities,: (i) *searchers* to implement ad-hoc searching strategies, (ii) *selectors* to bound only parts of the code in which we are interested, (iii) *analyzers* used to make per-path analysis of code.

The main advantage of symbolic execution is that **every bug found by the engine is "real"**. In other words, there are no false positives. Conversely, there may be false negatives because the symbolic execution engine can be unable to reach some paths in the code. Symbolic execution suffers of the well known *path explosion* problem. The number of paths that have to explored by the symbolic execution engine grows exponentially with the size of the symbolic input and the number of branches inside the code. However, with smart heuristics and selectors it is possible to effectively use symbolic execution in the verification of complex software systems.

1.3 Open Issues

Above we discussed that most of the device driver defects are related to protocols violations. Protocols are specifications of how the device driver, the OS resources and the device are expected to interact.

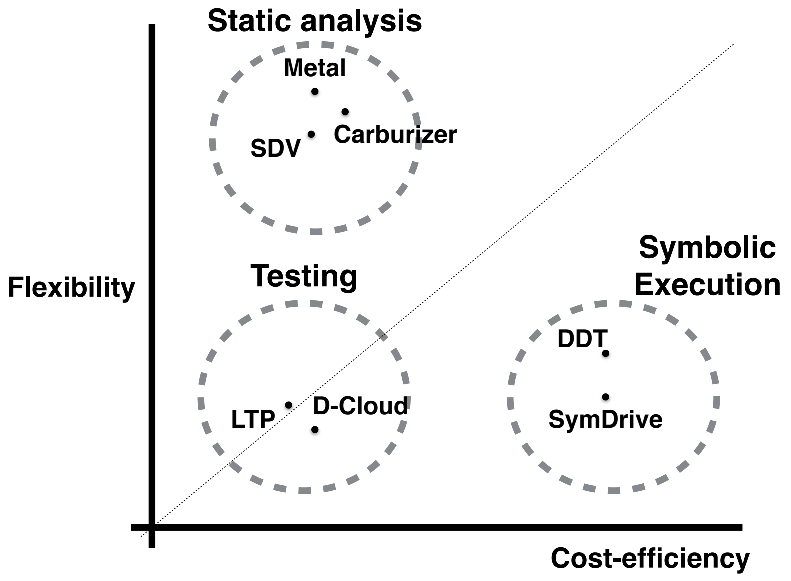


Figure 1.8: A synthesis of the trade-off between cost-effectiveness and flexibility of current approaches.

Figure 1.8 is a graphical illustration of the trade-off between cost-efficiency and flexibility of existing solutions. With *flexibility* we mean the capability of the tool of finding a wide set of defect types. On the other hand, *cost-efficiency* measures the capability of the approach of finding an high number of defects with a reasonable amount of human effort and of computational resources. Testing is not cost-efficient and flexible enough to find protocol bugs for the following reasons: (i) testing requires significant effort and resources ; (ii) testing is not able to detect a wide set of defect types because it frequently exercises the same portions of code. Symbolic execution performs better than testing because it allows the user to systematically explore paths in a program. Classic symbolic execution tries to explore all the possible paths in a program. Modern verification approaches based on symbolic execution significantly improve cost-efficiency because they focus the analysis only on "interesting" execution paths. However, the approaches based on symbolic execution are not flexible enough because they focus on generic programming defects such as memory leak, null pointer dereferences, race conditions since these bugs do not depend on a specific product e.g., a specific OS or a specific device. Static analysis approaches like SDV and Metal provide a high degree of flexibility, by allowing the user to customize the analyzer and, potentially, to find protocol bugs. However, static analysis is not cost-efficient enough because it requires a significant human effort for modeling the environment, e.g, OS API, device model.

1.4 Thesis Contribution

This thesis proposes the **model-based verification of device drivers** that allows to reach a better trade-off between cost-efficiency and flexibility. The approach enhances symbolic execution by enabling the user to control, using a model, the protocol that should be verified by symbolic execution. In this way, the user can adapt symbolic execution to defect types that apply to his own software, and can address protocol bugs that are not encompassed by traditional symbolic execution. In this way, the approach can achieve a great degree of flexibility, and at the same time can take advantage of the cost-efficiency of symbolic execution.

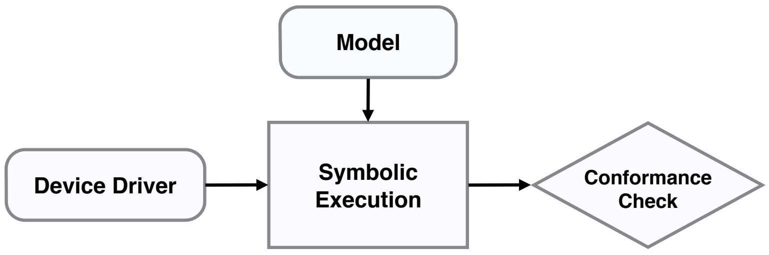


Figure 1.9: Overview of the model based verification approach for device drivers.

This approach consists in describing at high level of abstraction the properties that the device driver must hold and in verifying that the behavior of the driver complies with the property previously described. The idea is to compare the execution of the device driver with the model described at the first stage of the workflow. Divergence of the execution and the model along one of the possible sequence of instruction of the program determines a bug in the device driver. More specifically, the thesis makes the following contributions:

1. A methodology for *model-based verification* of device drivers through symbolic execution. In this context, this thesis proposes a language (SLANG) and an architecture (Symcheck). The language describes the behavior that the developer expects with respect to the API dependance, the hardware dependance. Chapter 2 provides the language specification and several examples. The architecture allows the **automatic**

transformation of the expected behavior specification, in a checker that exhaustively analyzes the conformance with the model.

2. Symbolic execution is a computationally intensive activity. Hence, new techniques are needed to improve the overall process. Thus, in the context of this thesis, I developed an enhanced symbolic execution platform that use a more efficient representation of intermediate code in order to speed up the symbolic execution of code. OS binary code is executed under a virtual machine (VM) environment and translated to the intermediate representation. Then, an interpreter symbolically executes the translated code. Tests on a pool of Windows device drivers showed that in this configuration we are able to symbolically execute more code e.g., 2x in the same amount of time respect with the current state of the art.

3. A methodology that aims at detecting protocol violations using behavioral models of device drivers. A protocol of device driver/device interaction is extracted from execution traces of device drivers under a set of workloads. The model of the protocol is then translated into a monitor that is attached at runtime to the device driver under test. If the device driver during its execution crosses state that do not belong to the model, the monitor signals that something unexpected happened.

1.5 Thesis Organization

The thesis is organized as follows. Chapter 2 presents a methodology to perform model-based verification of device drivers. Chapter 3 describes an approach to enhance symbolic execution of binary code. Chapter 4 describes a methodology for run-time verification of device drivers.

Chapter 2

Model-based Symbolic Execution for Device Drivers

2.1 Introduction

This chapter introduces Symcheck, a system that performs model-based verification of device drivers. Symcheck comes out with a language designed for simplifying the specification of protocols and a runtime support that orchestrates symbolic execution with model-based conformance checking. Section 2.2 introduces the Symcheck approach through a brief overview of the workflow.

2.2 Overview

This section provides a high-level overview of the approach. The overall approach aims at enabling developers to specify rules that must hold in device driver execution. The first part of the approach is SLANG, a language for specifying the logic of the protocol for the device driver. A protocol is *a specification of the expected interactions between the device driver,* *the OS kernel and the hardware device.*

A **checker** is a protocol written in SLANG. The SLANG checker is translated into a component that interacts with the symbolic execution engine, namely the **checker plugin**. The checker plugin is a component written in a general purpose language such as C++ or Java, that interfaces with the symbolic execution engine API. The developer formalizes the protocol in SLANG, a language tailored to model a wide range of protocols expressible through finite state machines.

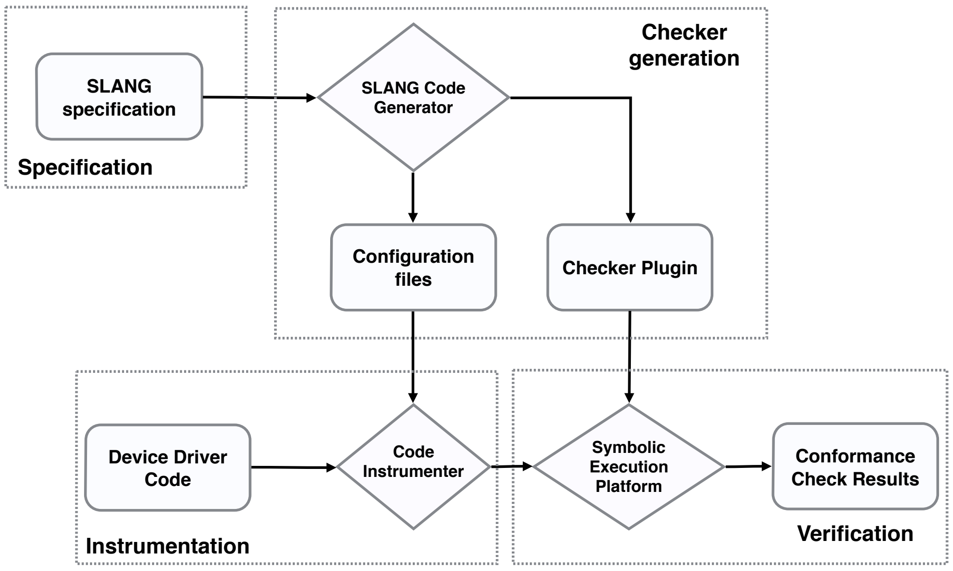


Figure 2.1: Symcheck workflow.

Figure 2.1 depicts the overall Symcheck workflow. The workflow consists of the following phases:

1. **Specification:** The developer specifies the protocol with SLANG language that abstracts low-level platform-specific API calls. We describe SLANG in section 2.3.

2. **Checker generation:** The specification is translated into the checker plugin that uses platform-specific API. The process is described in section 2.4. Checker generation produce two output files, (i) the plugin and (ii) a set of configuration files that feed both the instrumentation and the verification subsystems.

3. **Instrumentation:** An instrumentation tool uses configuration files produced in the previous phase to aid the verification phase.

4. **Verification:** Symbolic execution engine uses the checker plugin and the instrumented device driver to assess the conformance to the specification. This phase of the workflow is described in section 2.6.

2.3 Specification

SLANG provides the formalism to describe a protocol with a finite state machine semantics. The SLANG language has been defined to support the specification of events and states of device drivers' behavior. The events can be tied to the execution of specific APIs in the OS, and to accesses to OS key data structures. For instance, the device driver accesses to hardware registers and to OS events such as start of the interrupt handling routine, work-queues, spinlocks, etc., through an OS interface (such as the PCI or USB API): therefore, the language must allow to define events in terms of OS-specific APIs.

$\langle$*module*$\rangle$ ::= $\langle$*specification*$\rangle$;

$\langle$*specification*$\rangle$ ::= $\langle$*resource*$\rangle$ | $\langle$*checker*$\rangle$;

$\langle$*resource*$\rangle$ ::= **resource** $\langle$*name*$\rangle$ **{** $\langle$*member*$\rangle$ **};**

$\langle$*member*$\rangle$ ::= $\langle$*attribute*$\rangle$ | $\langle$*event*$\rangle$;

$\langle$*attribute*$\rangle$ ::= $\langle$*name*$\rangle$ **:** $\langle$*type*$\rangle$ **;**

$\langle$*type*$\rangle$ ::= **int** | **uint** | ... | **sym_expr**

$\langle$*event*$\rangle$ ::= **event** name **catches** signal $\langle$*where*$\rangle$

$\langle$*statement*$\rangle$ ::= expression **;** | $\langle$*symbolic_ val_ decl*$\rangle$

...

Figure 2.2: A part of the SLANG grammar.

2.3.1 The SLANG Language

The SLANG language aims at providing to the end-user a flexible way to specify protocols in the form of finite state machine (FSM). The grammar, shown in figure 2.2, formalizes SLANG.

Resource

Kernel components extensively use kernel facilities such as memory areas, semaphores, spin-locks. The kernel offers these objects to device drivers, which they use them to implement their functionalities such as allocate ring buffers, driver-specific data structures. A key aspect of protocol verification is the interaction between the device drivers with kernel resources. During execution, the device driver interacts with several instances of such resources, by accessing to them through APIs. Thus, we introduce in SLANG the concept of *resource*. Resources can be seen as "classes" in the sense of object-oriented languages: a resource specification describes the states of the resource from an abstract point of view; at run-time, many resources will be involved, and each resource will be represented by an "instance" of the general resource specification.

The user also defines the *events* that affect the resource, which can change the state of a resource instance. Events resemble "methods" of object-oriented languages. They are associated to the execution of key points in device drivers' code, such as the invocations of kernel APIs. When, at run-time, the driver executes a piece of code related to an event, the resource state will be updated accordingly. Events can be triggered when a kernel function or device-specific function (such as a read/write of a device register) is executed. Events can have zero or more input parameters, and zero or one output parameter. The parameters of an event are used in SLANG specification to update the state of the resource. Allocations and deallocations are special types of event, that model the creation and the deletion of a resource.

Figure 2.3 provides an example of resource specification. In this example, we model a dynamically-allocated memory area that is identified by its address. The allocation event *kmalloc* instantiates a new object. This object contains the address of the memory area (as the "val" member variable, whose definition is marked by the "id" keyword), which is obtained from the output parameter of the function call to *kmalloc*. When a *kmalloc* event occurs, a new instance of the *AddressSpace* resource is created. The "val" member variable represents the identifier of the resource instance. Similarly, *kfree* is the deallocation event. *Kfree* uses the “where” clause defined in SLANG to update the state of the specific memory area that is being deallocated (i.e., among all *AddressSpace* resource instances, only the instance related to the *kfree* event will be updated and deallocated).

```
resource AddressSpace {
    id val : uint32;

    //Events acting on the resource
    alloc event AddressSpaceAlloc catches Kmalloc {
        val = Kmalloc->ret;
    }

    dealloc event free catches AddressSpaceFree where (val == Kfree->arg0);

    //States definition
    state initial INIT{
        when (alloc) => ALLOCATED;
    };

    state ALLOCATED {
        when (free) => END;
    };

    state final END{
    };
};
```

Figure 2.3: Example of resource definition.

Finally, the last part of resource definition describes the finite state machine (FSM) model, that specifies how the resource reacts to events. Thus, the user formalizes with the FSM how a resource instance switches from a state to another one. Figure 2.3 shows the formalization of the kmalloc/kfree usage protocol using SLANG. A kernel memory area, for instance, can be allocated, used and finally deallocated. Each FSM has exactly one initial state. The user decides whether the FSM should have one or more final states, according to the protocol that he/she is modeling. Every state has one or more transitions towards other states. The FSM switches from one state to another state when a guard condition is satisfied. Guard conditions are expressed in form of boolean conditions, where the operands are the events previously defined in the resource declaration.

One important aspect of the language is the management of the resources identities, since several instances of resources can exist simultaneously during an execution. As mentioned above, we keep track of the identity of a resource using member variables (such as "val" in the example), and provide the *id* keyword to specifies which member variables are used to identify resources. When events are defined, we require the user to specify how to map the occurrence of an event to a specific resource instance. For instance, when a *Kfree* occurs, when want to update the state of the specific resource that is being deallocated (among the several resources that have been instantiated so far). So, the "where" clause must be used to specify the mapping between an event and a resource instance, by comparing the parameters of the event (inputs and outputs of the function call) to the member variables of the resource instance. Using this approach, we can keep track of events and states of each individual resource, and allow each resource to evolve independently.

Checker

SLANG checkers specify the protocol properties (over events and states of resource instances) that must be checked during the verification of device drivers. In other words, with checkers, a developer specifies combinations of states and events that should be considered *unexpected*. Figure 2.4 describes a simple protocol for the "AddressSpace" resource modeled in the previous section. User can add assertions that evaluate the state of resources, to ascertain the validity of a certain condition. The keyword *assert* specifies that the condition expressed in the body must be true when the execution meets a *verification point* (to be discussed later). Thus, if the condition is false, the checker detects a violation of the protocol.

```
checker KmallocProtocol {
    addresses : AddressSpace[];

    foreach a : addresses {
        assert a.currState == END;
    }
};
```

Figure 2.4: Example of checker definition.

The protocol described in figure 2.4 verifies that all instances of AddressSpace are in the END state (i.e., there are no leaks of dynamic memory).

Figure 2.5 shows a dynamic view of the algorithm 2.4. The figure depicts the symbolic execution tree and the state of every path that is symbolically executed. A state of symbolic execution includes the set of resources that have been instantiated so far during the execution of the path (e.g., AddressSpace instances that have been instantiated at a given time). Let suppose that in state $S1$ have been allocated N resources. The state of resources change depending on kernel events that are triggered when executing a certain state. Supposing that S1 executes a kmalloc, a new AddressSpace instance is created. The initial state of the new resource will be ALLOCATED. After that, the symbolic execution creates the state S3, in which $N + 1$ resources are allocated.

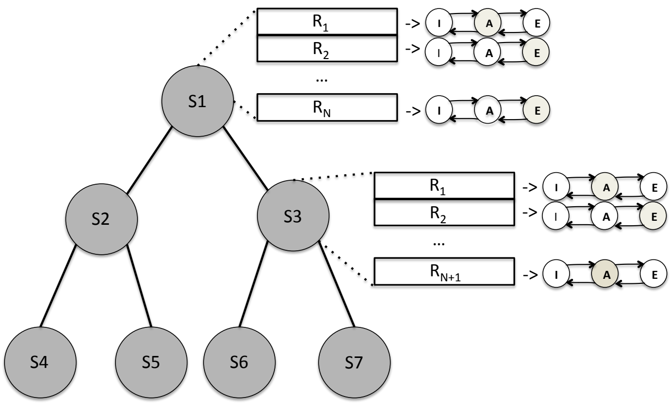


Figure 2.5: Dynamic view of the multipath execution with checker enabled.

Verification Points

The verification of protocols is performed during the execution of the device driver. In SLANG, the user can specify:

- The moment when the checker must be instantiated and enabled during execution. The checker plugin is enabled when the device driver code point specified by the user is executed (such as, the beginning of an I/O operation, or the initialization of the device driver).

- The moment when the checker must verify the validity of a protocol property. The checker plugin performs a control over the current execution state to check that the protocol property holds on a device driver code point specified by the user (such as, the end of an I/O operation or the deactivation of the device driver). If the protocol property is violated, the checker plugin reports a faulty execution.

The reason why we decided to demand to the user these aspects is because their definition changes across device drivers and protocol properties. In the case of the *kmalloc-kfree* protocol, for example, the lifecycle of a memory area should be within the lifecycle of the module that allocates and uses that memory area. In other cases, the lifecycle of the resource could be just one function, such as a function that performs an I/O operation.

```
checkpoints C {
    check KmallocProtocol at begin pcnet32.c:pcnet32_interrupt;
    check KmallocProtocol at end pcnet32.c:pcnet32_cleanup_module;
};
```

Figure 2.6: Example of testsuite definition.

The set of device driver code points in which we want to check the protocol is specified with the construct *checkpoints*. Each checkpoint contains a set of point declared with the keyword *check*. A check specifies both the name of the checker that we want to invoke, the point of the code where we want to enable the checker (keyword *begin*), and the point where we want to check the protocol (keyword *end*). The user can either specify these points in terms of function entries/exits, or in terms of specific statement (i.e., a line of code within a source code file) where the checker should be enabled and triggered.

Symbolic Source And Test Suite

To leverage symbolic execution, verification needs to inject symbolic values during the execution of device drivers. We opt to inject symbolic values at the interfaces of the device driver under verification: by using symbolic inputs at drivers' interfaces, we can exhaustively explore the execution paths within the code of the device driver, where the symbolic inputs represent the boundary conditions in which the driver operates. In particular, there are two driver interfaces that need to be considered: (i) the interface between the device driver and other kernel components (such as filesystem and the TCP/IP stack), and (ii) the interface between the device driver and the hardware. The former interface (towards the kernel) is a set of API functions exported by the device driver, and invoked by the OS to initiate an I/O transfer.

Figure 2.7 shows an example of interface for a network device driver in Linux, which exports functions for transmitting a network frame (e.g., hard_start_xmit), initializing the network interface card, gathering network statistics. The latter interface (towards the hardware) is represented by memory areas and I/O ports that must be read/written by the device driver to access to registers and buffers of the device controller. Memory locations are linked to device controller registers, so when there are memory operations (e.g., mov and load) the values are written directly into the device controller registers. For instance,

```
struct net_device
{
   char  name[ IFNAMSIZ ];
   int  ifindex;  /* inteface index */
   unsigned int mtu;  /* maximum transmission unit */
   unsigned char dev_addr[ MAX_ADDR_LEN ]; /* hardware address */
   void  *ip_pointer; /* points to IF?s IPv4 specific data */
   unsigned int flags;
   unsigned long trans_start; /* time (in jiffies) of last transmission */
   struct net_device_stats stats; /* a default set of device statistics */
   ...
   // some function-pointers (i.e., these are virtual functions)
   int (*open)( struct net_device * );
   int (*stop)( struct net_device * );
   int (*hard_start_xmit)( struct sk_buff *, struct net_device * );
   int (*get_stats)( struct net_device * );
   ...
};
```

Figure 2.7: Linux kernel structure net_device operations.

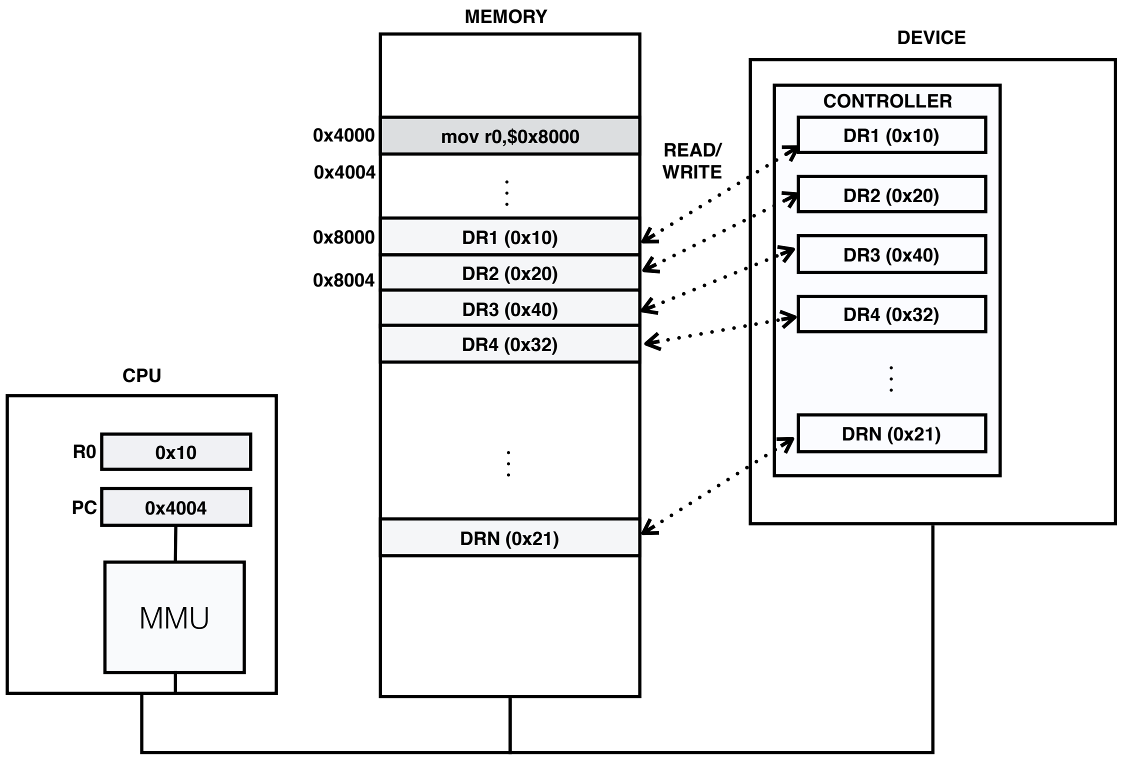


Figure 2.8: Memory mapped I/O.

figure 2.8 depicts how a device performs memory mapped I/O operations. Suppose that the CPU starts with a program counter $0x4000$. CPU fetches and decodes the instruction *mov* $r0, 0x8000$. Then when the CPU executes the *mov* instruction issues the address $0x8000$ on the bus. The bus circuitry enables the register $r0$ and queries for the value stored in $r0$. Finally, the value read from $DR1$ i.e., 0x10 is written into $R0$ and the CPU can update the program counter to the next word.

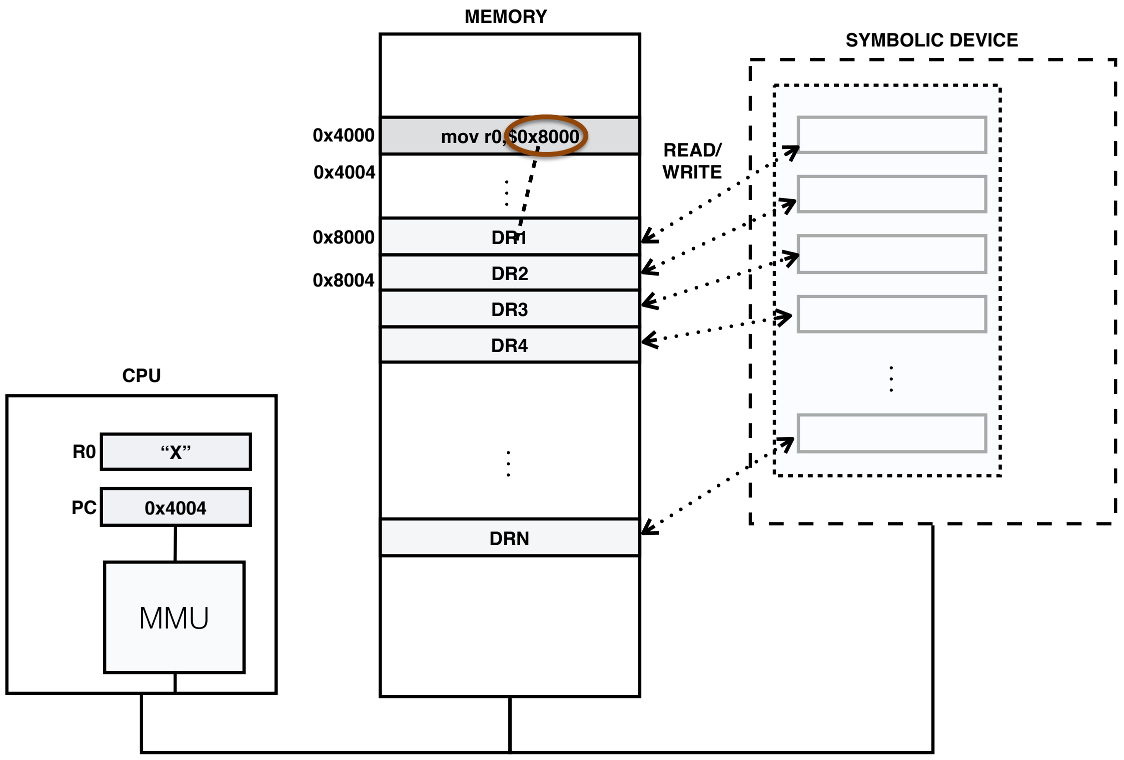


Figure 2.9: Memory mapped I/O and symbolic hardware.

To inject symbolic values at driver's interfaces, we adopt the following approach:

- To inject symbolic values at *driver invocations from the kernel*, the user specifies the entry points of the device driver in the SLANG language. In this way, the checker plugin will intercept the invocation of these entry points at run-time, and will replace the input parameters of the function calls with symbolic values. These symbolic values, in turn, will exercise the code paths within the device driver.

- To inject symbolic values *from the hardware device*, the checker plugin will intercept *read* operations to device registers and buffers, and will turn the data from the hardware into symbolic values. In this case, the user does not need to specify these sources of symbolic values, as they are automatically introduced by the *symbolic hardware* in the symbolic execution engine [16]. Symbolic hardware leverages memory mapped I/O mechanism used to support PCI or ISA devices to inject symbolic values at the device/driver interface. Figure 2.9 describes how symbolic hardware works. Let suppose that the CPU starts with a program counter $0x4000$. On instruction execution, the symbolic execution engine catches read at the memory address 0x8000 and writes a symbolic value in the $r0$ register. Whenever an instruction tries to read at a MMIO memory range the symbolic execution engine returns a symbolic value. Conversely, writes are discarded because on every read the symbolic hardware returns an unconstrained symbolic value.

The user specifies a device entry point in SLANG using a language keyword named *source*. With this keyword, the user selects where to inject a symbolic value, in terms of the location address (e.g., structure field, global variable) which must be executed.

```
testsuite T1 {
    source pcnet32_start_xmit {
        symbolic skb->len;
        symbolic skb->data;
    };

};
```

Figure 2.10: Example of testsuite definition.

Figure 2.10 shows an example of a test suite with two test cases for the driver pcnet32 in Linux. The keyword *testsuite* defines a set of symbolic sources. The name of the source is the entry point of the driver where we want to inject symbolic values. Each *source* contains a set of parameters that define which values are going to be marked as symbolic. In this example, the first source describes that when the pcnet32_start_xmit is called, the fields *len* and *data* of the packet parameter must be turned into symbolic data.

2.4 Checker Generation

Even if symbolic execution engines provide APIs to extend their functionalities, it is still a very onerous task to manually introduce a driver-specific protocol verification. For this reason, we automatically generate a tool (*checker plugin*) for extending a symbolic execution engine from the SLANG specification provided by the user. The automatically-generated checker plugin uses the APIs of the symbolic execution engine to inject symbolic values, and to verify protocol properties over the current state of driver's execution. More precisely, the *checker generator* translates the specification in a plugin for the target platform (i.e., the symbolic execution engine). The translator takes as input the file with the checker description in SLANG language. The code generator creates an *object tree*, which is a hierarchy of objects representing each entity in the SLANG specification (events, states, resources, ...). The generator parses the protocol specification, and then populates the object tree according to a metamodel of the SLANG language, following the general approach of domain-specific languages. We refer the interested reader to [34, 35] for further details about domain-specific languages. Checker generation produces in output a set of programs and configuration files that are used by the runtime support.

2.4.1 Translation of SLANG Resources

The most natural translation of a resource is into a class. An instance of the resource is created as a consequence of an allocation event. Listing 2.1 shows an example of translation of resource from the definition depicted in 2.3. As one can notice allocation events are translated into a factory method. Deallocation events are used in the finite state machine to decide when it is safe to deallocate the resource. The FSM manages resources lifecycle. On allocation event, the checker creates a new resource otherwise signals an error because the resource has been already allocated with the same id.

```
using boost::multi_index_container;
using namespace boost::multi_index;

//Stream to print messages
static llvm::raw_ostream* os;

```

```
class AddressSpaceResource {
  private:
    ref<Expr> m_val;


    AddressSpaceResource(uint32_t val) : m_val(val){}

  public:

    /***********************************/
    /*  Events declaration   */
    /*         */
    /***********************************/

    /***** Kmalloc *****/
    static boost::shared_ptr<AddressSpaceResource> EventKmalloc(S2EExecutionState* state,
         S2E_SYMINT_ARGS args){
      /* Statement compilation */
      uint32_t val = state->readMemory(args.ret,Expr::Int32);
      return boost::shared_ptr<AddressSpaceResource>(new AddressSpaceResource(val));
    }


    /***** Kfree *****/
    void EventKfree(S2EExecutionState* state,S2E_SYMINT_ARGS args){

    }

    bool whereKfreeCondition(S2EExecutionState* state,S2E_SYMINT_ARGS args){
      ref<Expr> var_0 = state->readMemory(args.arg0,Expr::Int32);
      return *m_val == *var_0;
    }

};

  //Initial states flags
struct FlagINIT {};

//Terminal states flags
struct FlagEND {};

//State machine
struct AddressSpaceDetector_ : public msm::front::state_machine_def<AddressSpaceDetector_>
{
  //State machine resources
  boost::shared_ptr<AddressSpaceResource> internal_ptr;

  //Constructor
  AddressSpaceDetector_(){}

  //States
  struct Init : public msm::front::state<>{
    template <class Event, class FSM>
    void on_entry(Event const&,FSM&) {
      *os << "Entering Init" << '\n';
    }

    template <class Event, class FSM>
  void on_exit(Event const&,FSM&) {
    *os << "Leaving Init" << '\n';
  }
};


//Initial state typedef
typedef Init initial_state;

struct Allocated : public msm::front::state<>{
  template <class Event, class FSM>
  void on_entry(Event const&,FSM&) {
    *os << "Entering Allocated" << '\n';
  }

  template <class Event, class FSM>
  void on_exit(Event const&,FSM&) {
    *os << "Leaving Allocated" << '\n';
  }
};

struct End : public msm::front::state<>{
  typedef mpl::vector1<FlagEND> flag_list;
  template <class Event, class FSM>
  void on_entry(Event const&,FSM&) {
    *os << "Entering End" << '\n';
  }

  template <class Event, class FSM>
  void on_exit(Event const&,FSM&) {
    *os << "Leaving End" << '\n';
  }
};


//Transitions
void __eventKmalloc(EventArgs const & ea){
    internal_ptr = AddressSpaceResource::EventKmalloc(ea.state,ea.args);
}
void __eventKfree(EventArgs const & ea){

}


//Guards
bool guardKmalloc(EventArgs const & evt){
    return true;
}

bool guardKfree(EventArgs const & ea){
  if(!internal_ptr->whereKfreeCondition(ea.state,ea.args)){
    *os << "Transition not possible" << '\n';
    return false;
  }
    return true;
}


```

```

  //Transition table
  typedef AddressSpaceDetector_ d;

  struct transition_table : mpl::vector<
  row<Init,EventArgs,Allocated,&d::__eventKmalloc,&d::guardKmalloc>,
  row<Allocated,EventArgs,End,&d::__eventKfree,&d::guardKfree>
  >{};

  //When a transition doesn't exist
  template <class FSM,class Event>
  void no_transition(Event const& e, FSM& fsm,int state) {
    *os << "no transition from state " << state
    << " on event " << typeid(e).name() << '\n';
  }
};

#endif
```

Listing 2.1: Code translated from SLANG to C++.

The translator generates two classes: one that models the resource (e.g., AddressSpaceResource), the other that model the finite state machine on the resource usage protocol (AddressSpaceDetector_). An allocation event the detector creates a new instance of the resource AddressSpace. When the AddressSpaceDetector receives an event, if a guard is triggered, the detector changes the state of the resource. All the feasible transitions are stored into a transition table which contains also the guards that trigger the state change.

2.4.2 Translation of SLANG Checkers

Checkers are translated into procedure that access to the resources dynamically allocated during the system's evolution. Despite the case of canonical programming languages, the line 2 in code snippet in figure 2.4 means that we access at runtime into an array of AddressSpace elements. The procedure depicted in figure 2.2 shows an example of the specification translation from SLANG to C++ .

```
for(DetectorsByAddr::iterator it = byAddr.begin(),
     ite = byAddr.end(); it != ite; it++){
     PciMapWrapper wp = (*it);
     bool isTerminated = wp.det.is_flag_active<FlagEND>();
     if(isTerminated){
         DEBUG_S() << "[PciMap] Everything seems fine on path_id " << state->getID() << "\n
             ";
     }else{
         WARNING_S() << "[PciMap] There is at least one protocol violation on path_id " << state
             ->getID() << '\n';
         DEBUG_S() << "[PciMap] Finding an example that triggers this protocol violation..." << '\n
             ';
         s2e()->getExecutor()->terminateStateEarly(*state,"PciMap");
     }
    }
```

Listing 2.2: Snippet of checker specification translation for the DMA map protocol.

The iteration over AddressSpace elements becomes a loop in C++ over the AddressSpace resource instances. Then, the assert statement is translated into a check on the current AddressSpace instance state. If the check fails then the procedure emits an error string and terminates the current state.

2.4.3 Translation of SLANG Testsuites

Testsuites are converted into a configuration file that is used during the code instrumentation phase. Let consider testsuite defined in section 2.10, that model a testsuite for the functions pcnet32_start_xmit.

```

T1 - [ pcnet32_start_xmit : skb->len , skb->data]
```

Figure 2.11: Testsuite translated from SLANG to a configuration file used for instrumentation.

In this case, the checker generator produces for this input the output shown in 2.11. The configuration file can contain sets of testsuite declarations that consists of testsuite name and square brackets. Each configuration line contains the name of the target function and the names of the arguments that we want mark as symbolic.

2.4.4 Translation of SLANG Verification Points

As we mentioned in section 2.3.1, verification points are selected through the *checkpoints* keywords. Similarly to testsuite generation the checker generator produces a configuration file that encodes the points in which the instrumented should insert the call to the protocol checker. Referring to the figure 2.6, the checker generator translates the checkpoints producing the configuration file shown in figure 2.12.

```
C − [ b@pcnet32_interrupt : KmallocProtocol; e@pcnet32_cleanup_module : KmallocProtocol; ]
```

Figure 2.12: Verification points translated from SLANG to a configuration file used for instrumentation.

2.5 Instrumentation

This section describes how we perform instrumentation of device drivers. Configuration files establish which entry points the instrumentation tool must instrument according to the SLANG specification. Instrumentation can be performed modifying the source code or introspecting the virtual machine state at runtime. Introspection consists in inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it [36]. However, since we assume that device driver code is available for testing and verification purposed we choose a code rewriting approach. Thus, the code rewriter tool during

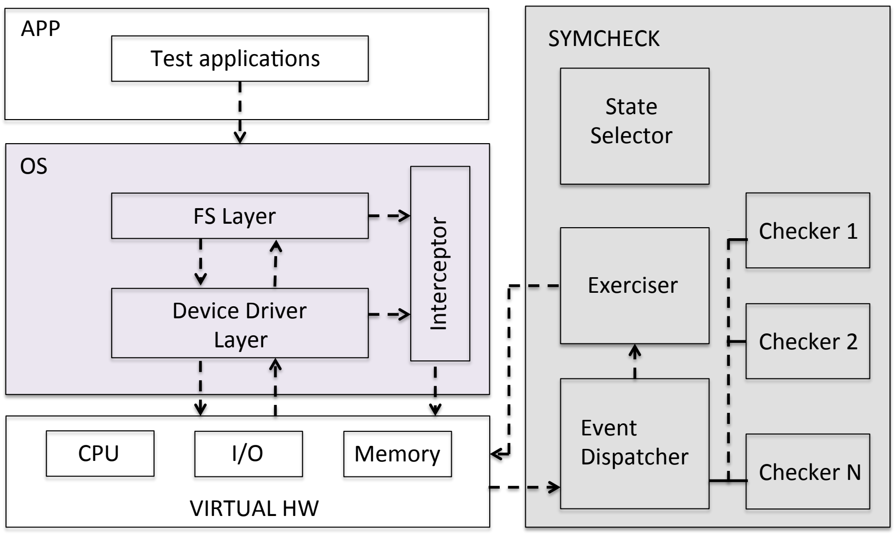


Figure 2.13: Backend architecture.

the analysis uses the informations provided by the configuration files to insert call to the interceptor module, explained in section 2.6.1. The instrumented device driver code aids the verification phase: (i) guiding symbolic execution, (ii) inserting calls to the interceptor. Figure 2.20 depicts the code instrumented with checks. In the instrumentation phase, the system inserts call at the code points defined in specification.

2.6 Verification

This section describes the verification architecture that uses the generated checker plugin during symbolic execution. The design of the architecture is showed in figure 2.14. In the following, we discuss in more detail each individual component of this architecture.

2.6.1 Interceptor

The operating system, together with the device driver under verification and with user-space applications, are executed on a virtual machine. User applications executes on top of the OS kernel, and they interact with it using system calls. Since events related to verification are associated with kernel functions, we need a mechanism to map such functions with events. The most natural approach is to statically instrument the driver code: every call to interested function is rewritten as a function calls to the *interceptor layer* (IL) within the symbolic execution engine. This invocation is performed by instrumenting the driver's code with special opcodes (i.e., new machine instructions not already used by the emulated hardware architecture) that are intercepted and forwarded by the virtual machine. The interceptor layer is the part of the system preposed to collect such "interesting" events, that is, the set of functions and instructions that are specified by the user in SLANG for protocol verification purposes.

2.6.2 Event Dispatcher

When the Interceptor issues an event, it is collected and dispatched by the *event dispatcher* (ED) component. It sends signals to the checker object: for instance, a call to *kmalloc* in the device under verification is translated into an internal *kmalloc* signal in the checker plugin, which is dispatched to the checker object that verifies the *kmalloc/kfree* protocol. Each checker subscribes to events of interest, and receives a signal when the event occurs during the execution of a device driver. For instance, a checker interested in the *kmalloc* event subscribes to the dispatcher *Kmalloc* event.

2.6.3 Exerciser

To perform symbolic execution of the driver's code, we need to inject symbolic values at specific points of the component interface that we want to test. The *Exerciser* injects symbolic values at the kernel/driver interface and at driver/device interface. The Exerciser consists in a code rewriting tool that instruments the points of the code starting from the configuration file generated in the phase described in 2.3.1. Thus, the exerciser injects the call to the function that makes symbolic value directly into the code. We opted to let the user choose, through the use of symbolic source 2.3.1, the point of the code in which he/she want to inject symbolic values, to give a high flexibility in the creation of test suites. The Exerciser interacts with the symbolic execution engine to introduce a symbolic variable in the current state of the execution: from that point on, every access to that symbolic variable will trigger a fork of the current execution state, following the general approach of symbolic execution (see also section 1.2.3).

2.6.4 Symbolic Execution Scheduler

In general, symbolic execution leads to the exploration of a huge number of paths, where each of them consumes computational resources such as CPU and memory. In almost all real-world software systems (device drivers are not an exception on this regard), it is not feasible to exhaustively explore all these paths. For this reason, there is a need for *heuristic rules* that focus the symbolic execution only on the most important and "error-prone" paths of the driver under verification. An heuristic selects which path (among all the paths viable at a given time) must be executed next, and has a tremendous impact on the effectiveness of symbolic execution. Branch heavy code (e.g., chip selection code that chooses the right chip in a probe function), and polling loops, are among the main sources of the state explosion problems for device drivers. State explosion for loops and initialization code has been widely explored in [15] and [16]. Our Symcheck approach inherits from Symdrive the heuristics for handling polling loops and device initialization code.

For polling loop handling, Symcheck implements *loop elision*, which gives priority to paths that quickly exit the loop. For instance, suppose an execution path E, when E reaches a loop instruction the symbolic execution engine will fork two paths the first that enters the loop and the second that exits the loop. Loop elision tries to select the path that exits earlier the the loop. The strategy is greedy, at each iteration of the loop the path that forks new paths is deprioritized. If there is a complex chain of loops the approach randomly selects a path with the hope of exiting the loop. An alternative strategy, adopted by DDT, is to introduce a plugin that kills edges in the binary control flow graph. However, this solution is too cumbersome since needs to explicitly mark the binary offset where the victim edge is located, but those offsets change at each recompilation of the device driver and need to be reconfigured at each run. To further improve the effectiveness of these heuristics, these user can annotate driver code to give lower priority to paths in which symbolic execution execution gets stuck, such as algorithms that compute checksums.

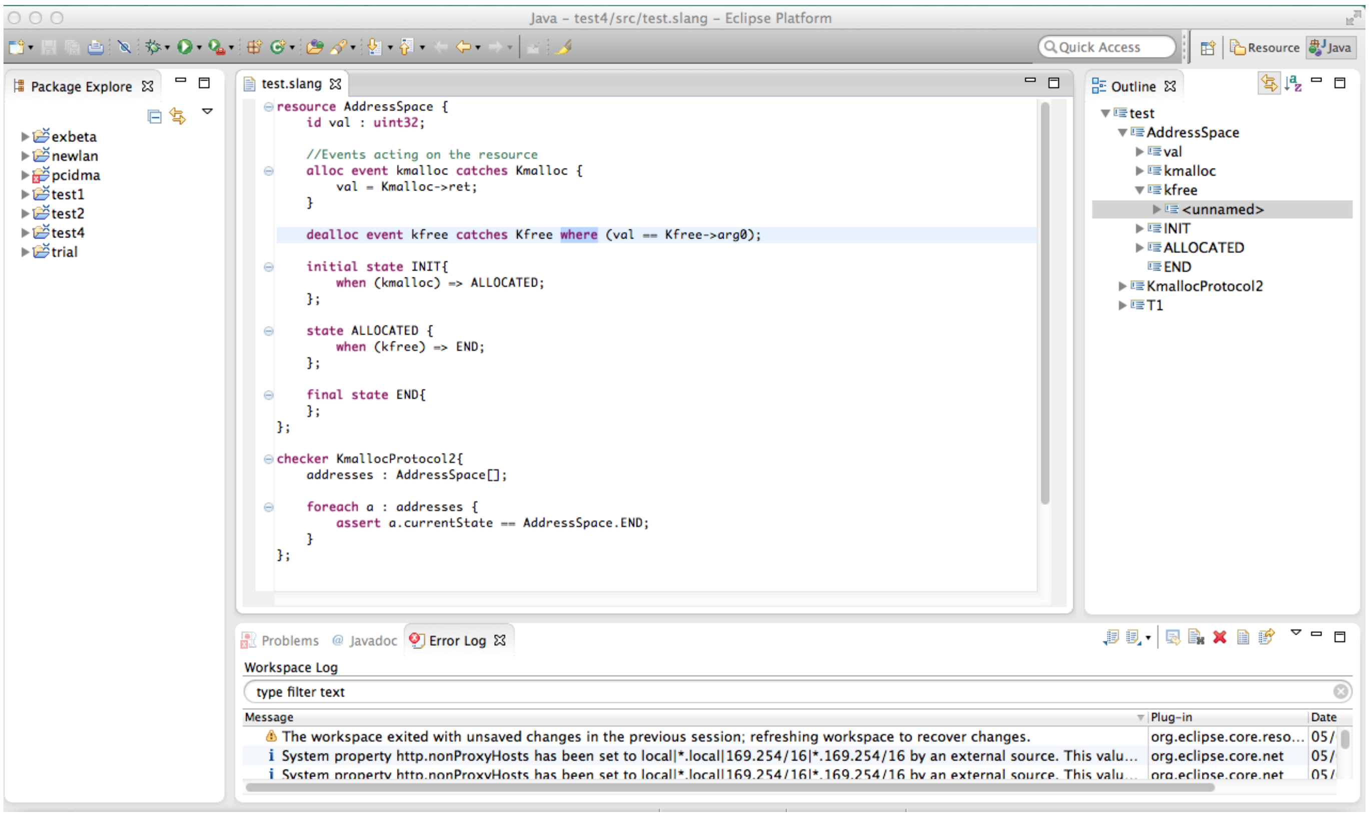
To address path explosion problem due to branch heavy code, we used the *favor-success* heuristic. Symbolically execute device driver initialization code entails path explosion because the code often has many branches to support multiple chips and configurations. Favor-success prioritizes paths that do not return errors. A simple way is to kill paths that correspond to hardware error configuration. Since we are interested in verifying the driver only when the device is correctly configured, this approach enforce the symbolic execution engine to execute in interesting portions of device driver code (e.g., network packet send, block read).

2.7 Implementation Details

This section describes the implementation details of Symcheck. In section 2.7.1 we describe the implementation details regarding the SLANG whereas in the section 2.7.2 we cover more deeply how we implemented the backend.

2.7.1 SLANG Implementation

The front-end is implemented using the Xtext infrastructure in the Xtend programming language. Xtext [37] is a framework for the development of programming languages and domain specific languages. It handles all the aspects of a complete language infrastructure process such as parsing, compiling and linking and interpretation. Xtext allows to describe the grammar of the language in EBNF form. The Xtext engine uses this grammar to automatically generate a parser for the language and an infrastructure used for add language-specific mechanism such as scoping and type checking and code generation.



```
resource AddressSpace {
    id val : uint32;

    //Events acting on the resource
    alloc event kmalloc catches Kmalloc {
        val = Kmalloc->ret;
    }

    dealloc event kfree catches Kfree where (val == Kfree->arg0);

    initial state INIT{
        when (kmalloc) => ALLOCATED;
    };

    state ALLOCATED {
        when (kfree) => END;
    };

    final state END{
    };
};

checker KmallocProtocol2{
    addresses : AddressSpace[];

    foreach a : addresses {
        assert a.currentState == AddressSpace.END;
    }
};
```

```
Message                                                                                          Plug-in                   Date
⚠ The workspace exited with unsaved changes in the previous session; refreshing workspace to recover changes.   org.eclipse.core.reso...  05/
i System property http.nonProxyHosts has been set to local|*.local|169.254/16|*.169.254/16 by an external source. This valu...   org.eclipse.core.net   05/
i System property http.nonProxyHosts has been set to local|*.local|169.254/16|*.169.254/16 by an external source. This valu...   org.eclipse.core.net   05/
```

Figure 2.14: A screenshot of the SLANG editor for specifying the language.

The code generator translates the checker expressed in SLANG into C++. The SLANG state machines are translated into C++ using the Boost meta-state machine library [38]. A translated SLANG checker interfaces with S2E plugin API to inject symbolic value and to observe events. It is straightforward adding the support for the API of a different symbolic execution engine such as KLEE. It requires only the modification of the platform-specific API invocations.

2.7.2 Runtime Support Implementation

Conversely, the back-end is built in C++ on top of S2E. Symcheck uses the S2E plugin infrastructure to provide functionalities. A first plugin named “LinuxKmInterceptor” intercepts some kernel events such as module loading and module unloading, those events are useful to start the monitoring. We require only a modification to the kernel function that loads the kernel module (i.e., *load_module*). The kernel send the name of the loaded driver to the plugin that parses the ELF (Executable and Linkable Format) sections of the file itself. From these sections, the plugin creates a module object that is dynamically added to the pool of modules tracked by S2E. Then, the plugin emits a signal that notifies that a kernel module has been loaded to other plugins. A kernel module implements the interceptor layer that is named *symmod*. The instrumented device driver under test invokes the entry points of the interceptor layer, that forwards the call to internal functions of the Symcheck guest API.

The guest API populates the message described in figure 2.18 with the addresses of the arguments and of the return value. Then, plugins parse this data structure to read concrete/symbolic values that are stored in memory. Symcheck executes special opcodes x86 to communicate with the host-side infrastructure. The “SymCheckDispatcher” plugin is responsible of dispatching kernel events, sent by the interceptor, towards checkers. This plugin accepts messages sent from the guest side through the Symcheck guest API. Checkers react to events dispatched by the “SymCheckDispatcher” plugin. When a checker receives an “interesting” event it changes the internal state according to the model specified in SLANG. *CheckProtocol* is a special event that is triggered when the code execution reaches the points specified using the construct *checkpoints.*

The “Exerciser” is a source code rewriter written in C++ built with the Clang framework [39]. Clang is an open-source compiler for the C family of programming languages. Clang is built upon the LLVM infrastructure. LLVM offers a versatile framework to design a broad class of analysis, such as source code transformations, code optimization, static analysis etc.

There are many projects built with LLVM and Clang [40] with tools targeted for software verification [41, 27] and enforcing run-time safety rules such as API integrity enforcement [42] , memory safety [43], deterministic execution [44, 45]. The “Exerciser” takes the configuration file generated by the frontend and it injects code in the points of the device driver code defined in the configuration file.

2.8 Evaluation

This section describes the results of applying model-based symbolic execution of device drivers on a real Linux device driver. The following paragraphs show two cases of defect types. The section 2.8.1 analyzes a case of a DMA resource leak while the section 2.8.2 shows an example of a detector of device driver/device data race.

2.8.1 Case 1: Resource Leak

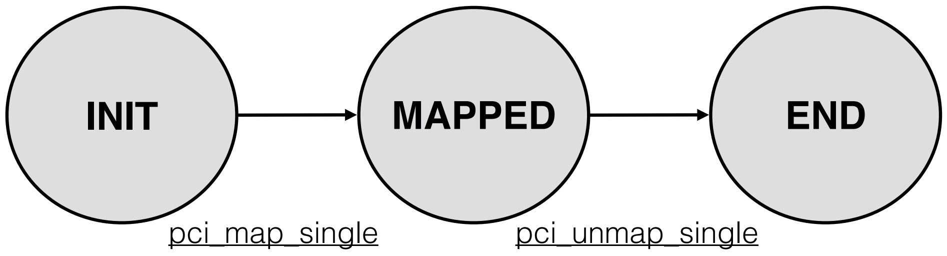


Figure 2.15: State machine of a DMA memory address space.

The case analyzed in this section regards resource leaks of DMA address on PCI device drivers . According to the documentation [20] every call of the function pci_map_single must be followed by a call to the pci_unmap_single. The presence of a protocol violation can cause incorrect use of bus addresses, when the bus address are exhausted they may cause unexpected behavior. Figure 2.15 describes a finite state machine of the DMA mapping protocol. We model each DMA memory area as a finite state machine that can transit in the following states: (i) INIT; (ii) MAPPED; (iii) END.

When a dynamic memory area owned by the driver is mapped into a bus address through the pci_map_single function, the finite state machine transits from the INIT state towards the MAPPED state. Whenever the device driver calls the pci_unmap_single o previously mapped area the finite state machine transits in the END state. The finite state machine signals the user of violation when the device driver send events that are not present in the model. Figure 2.19 represents a bug that we found with Symcheck that pcnet32 device drivers. We configured as protocol verification points the end of the send and the kernel module deallocation points. The analysis showed that areas allocated by the pci_map_single function are not deallocated neither in the interrupt handler nor when the kernel cleans up the module.

2.8.2 Case 2: Data Race

In this section, we describe an example of data race between device driver and device. Device drivers manipulate DMA buffers using both virtual and bus addresses. For example, the virtual address is used to read/write the DMA buffer, while the bus address is used to synchronize the cache and memory copies of a DMA buffer to avoid coherence issues (e.g., dma_sync_single for cpu). An example of data race during a DMA operation is depicted in figure 2.16 [46]. The device driver at one point of its execution maps buffers to DMA transfer area. Then, when the device driver does not need the DMA area anymore executes an unmap. During the interval between a map and the next unmap , only the device can access the mapped area in order to asynchronously perform the DMA transfer from memory to the device. Thus, an eventual write of the device driver on the mapped area could generate races.

The state machine describing the protocol is depicted in figure 2.17. In this case, each

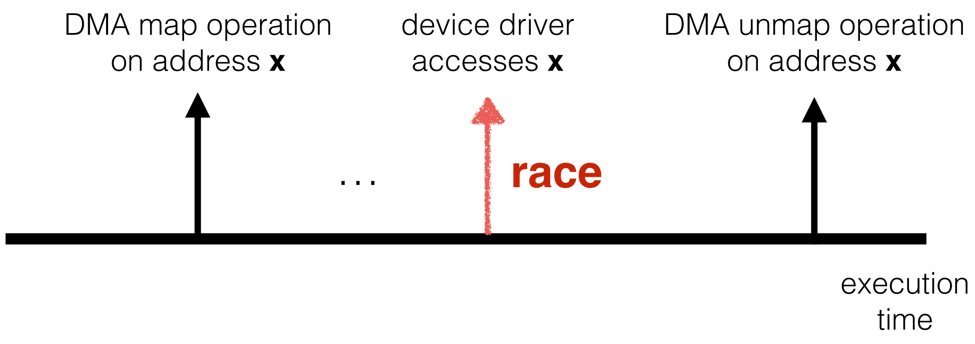

DMA map operation on address **x**
device driver accesses **x**
DMA unmap operation on address **x**
...
race
execution time

Figure 2.16: Example of a data race between device driver and device.

DMA memory are can transit in each of the following states: (i) INIT; (ii) MAPPED; (iii) END; (iv) RACE. The transition INIT to MAPPED happens when the device driver performs a call to the *pci_map_single* function. In this case, we have two final states: the END state represents a correct unmap i.e., the device driver invokes *pci_unmap_single* when it does not need anymore the DMA memory area. The RACE state represents the condition when the device driver writes data to the DMA memory mapped I/O area before that the device driver unmaps it. The checker reacts if the device driver violates the protocol, writing on a memory area that is still mapped.

We performed experiments using this checker on the Linux device driver pcnet32. In order to force complex interleavings we injected interrupts randomly. Injecting interrupts randomly may lead in the generation of spurious interrupts, i.e., interrupts that a functioning device cannot issue. However, the device driver must be tolerant to situations in which the device is faulty.

In order to test the correct behavior of the checker we injected a fault in the function

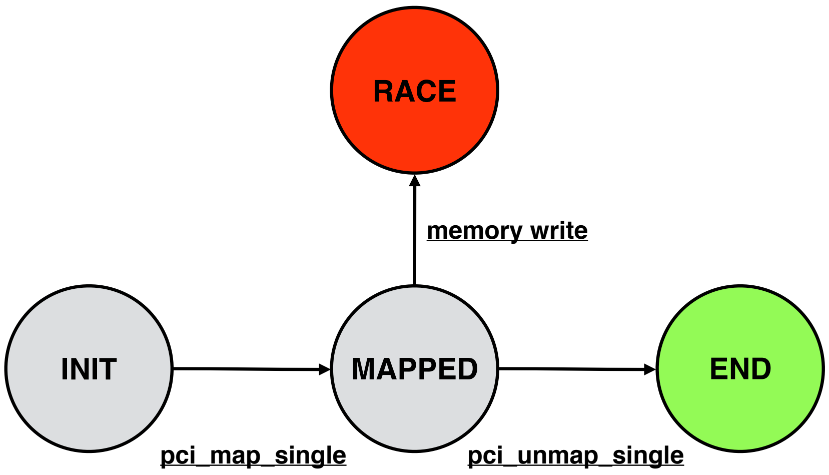


Figure 2.17: State machine of the DMA data race detector.

pcnet32_purge_rx_ring as shown in figure 2.21. In this case, the device driver did not unmap the area that belongs to the ring buffer. The checker detected a protocol violation when the operating system tried to reuse the mapped area. Furthermore, we were able to reconstruct the interactions on the use of DMA buffers between the OS and the device driver.

```

struct S2E_SYMINT_ARGS{
    uint32_t arg0;
    uint32_t arg1;
    uint32_t arg2;
    uint32_t arg3;
    uint32_t ret;
} __attribute__((aligned(8)));;

struct S2E_SYMINT_COMMAND {
    S2E_SYMINT_COMMANDS Command;
    union {
        S2E_SYMINT_ARGS args;
    };
} __attribute__((aligned(8)));;
```

Figure 2.18: Message format to communicate events to the dispatcher.

```
static netdev_tx_t pcnet32_start_xmit(struct sk_buff *skb,
              struct net_device *dev)
{
  ...

  /* Caution: the write order is important here, set the status
   * with the "ownership" bits last. */

  lp->tx_ring[entry].length = cpu_to_le16(-skb->len);

  lp->tx_ring[entry].misc = 0x00000000;

  lp->tx_skbuff[entry] = skb;
  lp->tx_dma_addr[entry] =
      pci_map_single(lp->pci_dev, skb->data, skb->len, PCI_DMA_TODEVICE);
  lp->tx_ring[entry].base = cpu_to_le32(lp->tx_dma_addr[entry]);
  wmb();  /* Make sure owner changes after all others are visible */
  lp->tx_ring[entry].status = cpu_to_le16(status);

  lp->cur_tx++;
  dev->stats.tx_bytes += skb->len;

  /* Trigger an immediate send poll. */
  lp->a.write_csr(ioaddr, CSR0, CSR0_INTEN | CSR0_TXPOLL);

  dev->trans_start = jiffies;

  if (lp->tx_ring[(entry + 1) & lp->tx_mod_mask].base != 0) {
    lp->tx_full = 1;
    netif_stop_queue(dev);
  }
  spin_unlock_irqrestore(&lp->lock, flags);
  return NETDEV_TX_OK;
}
```

Figure 2.19: Example of a DMA resource leak found in Linux pcnet32 device driver.

```
/* The PCNET32 interrupt handler. */
static irqreturn_t pcnet32_interrupt(int irq, void *dev_id) {
  symmod_check_protocol("KmallocProtocol");
        struct net_device *dev = dev_id;
        struct pcnet32_private *lp;
        unsigned long ioaddr;
        u16 csr0;
        ...
        spin_lock(&lp->lock);

        csr0 = lp->a->read_csr(ioaddr, CSR0);
        while ((csr0 & 0x8f00) && --boguscnt >= 0) {
                if (csr0 == 0xffff)
                        break; /* PCMCIA remove happened */
                /* Acknowledge all of the current interrupt sources ASAP. */
                lp->a->write_csr(ioaddr, CSR0, csr0 & ~0x004f);

                netif_printk(lp, intr, KERN_DEBUG, dev,
                          "interrupt csr0=%#2.2x new csr=%#2.2x\n",
                          csr0, lp->a->read_csr(ioaddr, CSR0));

                /* Log misc errors. */
                if (csr0 & 0x4000)
                        dev->stats.tx_errors++; /* Tx babble. */


    ...
        return IRQ_HANDLED;
}


static netdev_tx_t pcnet32_start_xmit(struct sk_buff *skb, struct net_device *dev) {
        struct pcnet32_private *lp = netdev_priv(dev);
        unsigned long ioaddr = dev->base_addr;
        u16 status;
        int entry;
        unsigned long flags;

        spin_lock_irqsave(&lp->lock, flags);

        netif_printk(lp, tx_queued, KERN_DEBUG, dev,
                    "%s() called, csr0 %4.4x\n",
                    __func__, lp->a->read_csr(ioaddr, CSR0));

  ...

        lp->tx_ring[entry].length = cpu_to_le16(-skb->len);

        lp->tx_ring[entry].misc = 0x00000000;

        lp->tx_dma_addr[entry] =
            pci_map_single(lp->pci_dev, skb->data, skb->len, PCI_DMA_TODEVICE);
        if (pci_dma_mapping_error(lp->pci_dev, lp->tx_dma_addr[entry])) {
                dev_kfree_skb_any(skb);
                dev->stats.tx_dropped++;
                goto drop_packet;
        }
        lp->tx_skbuff[entry] = skb;
        lp->tx_ring[entry].base = cpu_to_le32(lp->tx_dma_addr[entry]);
        wmb(); /* Make sure owner changes after all others are visible */
        lp->tx_ring[entry].status = cpu_to_le16(status);
  ...

 drop_packet:
        spin_unlock_irqrestore(&lp->lock, flags);
  symmod_check_protocol("KmallocProtocol");
        return NETDEV_TX_OK;
}
```

Figure 2.20: Device driver code instrumented with verification points

```
static void pcnet32_purge_rx_ring(struct net_device *dev)
{
        struct pcnet32_private *lp = netdev_priv(dev);
        int i;

        /* free all allocated skbuffs */
        for (/*i = 0*/ i = 1; i < lp->rx_ring_size; i++) {*/
                lp->rx_ring[i].status = 0; /* CPU owns buffer */
                wmb(); /* Make sure adapter sees owner change */
                if (lp->rx_skbuff[i]) {
                        s2e_printk("----DMA unmap @ line %d\n",__LINE__);
                        symmod_pci_unmap_single(lp->pci_dev, lp->rx_dma_addr[i],
                                        PKT_BUF_SIZE, PCI_DMA_FROMDEVICE);
                        dev_kfree_skb_any(lp->rx_skbuff[i]);
                }
                lp->rx_skbuff[i] = NULL;
                lp->rx_dma_addr[i] = 0;
        }
}
```

Figure 2.21: Fault injected in the Linux AMD pcnet32 driver.

Chapter 3

Enhancing Performance of Symbolic Execution

This chapter describes an enhanced symbolic execution platform with emphasis on performance, in terms of speed of program state-space exploration. The enhanced platform addresses a key performance bottleneck in symbolic execution: translating a binary program (encoded using a "source ISA") into another representation ("target ISA") amenable to symbolic execution. The enhanced platform introduces a translation-efficient code representation that improves the performance of the symbolic execution engine. Before introducing the enhanced platform, we give an introduction about how a symbolic execution engine internally works, in order to understand the phases that impact significantly on symbolic execution performance.

3.1 Background

In this section, we briefly recap the basic concepts on binary translation and interpretation that are required to understand how symbolic execution engines work.

3.1.1 Interpretation

Interpretation is a technique that mimics the behavior of a CPU. An interpreter fetches, analyzes and executes a set of instructions in a loop. In the history of computer science, interpretation was born very early. Popular languages such as LISP, Perl and Java were based on interpretation. This section gives a brief overview about how interpretation techniques are applied to binaries. An interpreter operates on a in-memory representation of the source machine state. Figure 3.1 depicts the structure of a generic interpreter. The address

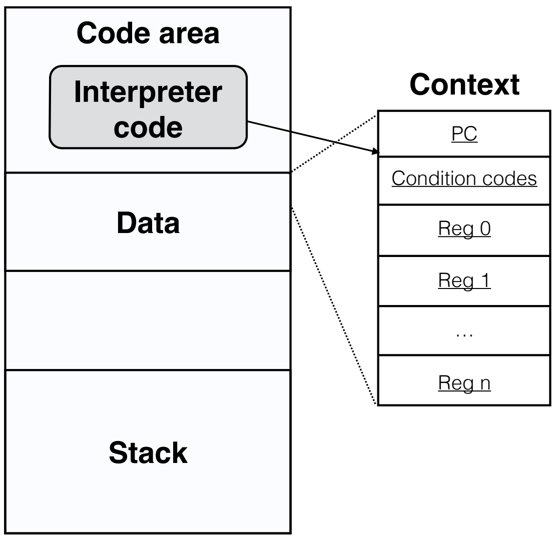


Figure 3.1: Interpreter structure.

space in which is stored the interpreter also contains a table referred as *context block*, that maintains the various components of the source machine state, such as general-purpose registers, the program counter, condition codes, and additional control registers.

```
while (!halt && !interrupt) {
     inst = code[PC];
     opcode = extract(inst,31,6);
     switch(opcode) {
          case LoadWordAndZero: LoadWordAndZero(inst);
          case ALU: ALU(inst);
          case Branch: Branch(inst);
          . . .}
}

Instruction function list

LoadWordAndZero(inst){
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     displacement = extract(inst,15,16);
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data[address]<< 32) >> 32;
     PC = PC + 4;
}

ALU(inst){
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     RB = extract(inst,15,5);
     source1 = regs[RA];
     source2 = regs[RB];
     extended_opcode = extract(inst,10,10);
     switch(extended_opcode) {
          case Add: Add(inst);
          case AddCarrying: AddCarrying(inst);
          case AddExtended: AddExtended(inst);
          . . .}
     PC = PC + 4;
}
```

Figure 3.2: Interpreter loop for a PowerPC architecture.

The interpreter executes the source binary instruction by instruction, the instructions can read and modify the state stored in the context block according to their semantics . This kind of interpreter is referred as *decode-and-dispatch* interpreter, because there is a main loop that decodes an instruction and then dispatches it to an handler that mimics its behavior. Figure 3.2, extracted from [47], shows a decode-and-dispatch interpreter for a PowerPC architecture. The main interpreter loop starts fetching an instruction from memory ,at the address *code*[*PC*], and then it extracts its opcode. The switch construct selects based on the opcode which operation must be performed. The Load Word and Zero instruction loads a 32-bit word in a 64-bit register and zeroes the most significant registers. Conversely, the set of instructions that exercise the ALU emulate all the arithmetic and logic operations such as addition, subtraction, multiplication etc. At the end of each instruction handler the program counter is incremented by 4, that is the instruction size. An approach to make interpretation faster is to let the common case as fast as possible.

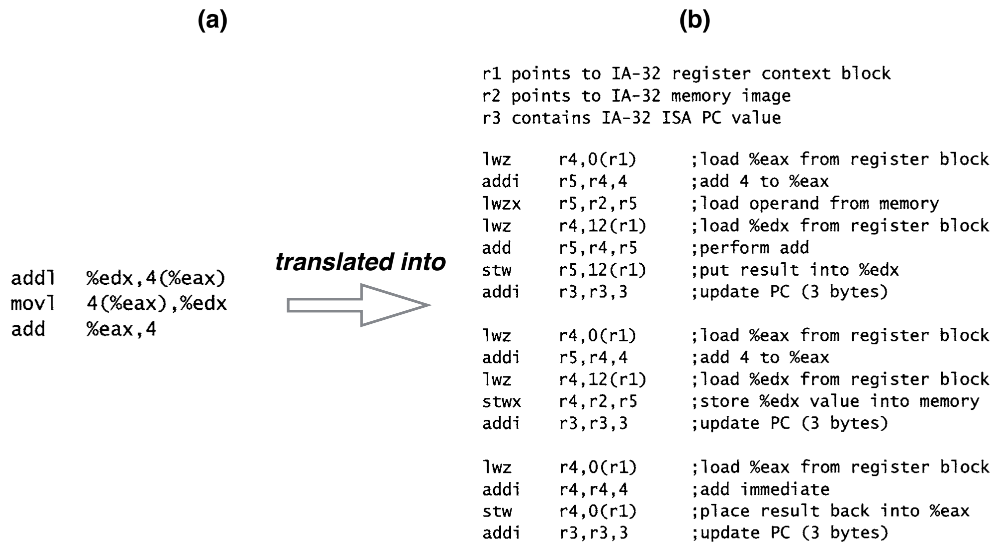
3.1.2 Dynamic Binary Translation

Interpretation suffers of performance overhead. The issue is that a single source ISA operation costs up to thousands of target CPU cycles. The mapping each individual source instruction to its own customized target code minimizes the number of target ISA instructions needed to emulate a source ISA instruction. This process of converting a source binary into a target binary is known as *binary translation.* A binary translator divides the source instruction stream in instruction chunks named *basic blocks.* Basic blocks contains a sequence of instructions where the last instruction is an operation that modifies the control-flow, such as branch, a call or a return.

A static binary translator translates source ISA binary code to target ISA binary code without having to run the code. On the other hand, dynamic binary translation [47] selects and executes basic blocks during the code execution. This section focuses on dynamic binary translation. Dynamic binary translation has been used for several purposes such as virtualization [48], testing/verification [49], debugging [50], profiling [51], sandboxing [52]. A dynamic binary translator can be imagined as a program that performs the following steps continuously (interrupt handling is omitted in this description):

1. fetch instructions from an instruction stream
2. translate the basic blocks from source ISA to the target ISA
3. execute the basic block on the execution engine either by direct execution on the host CPU or by interpretation.
4. return to step 1

Regarding step 3, it must be noticed that symbolic execution takes advantage of both direct execution and interpretation. Direct execution is adopted to quickly execute, on the physical processor, the parts of a program that do not involve symbolic execution, thus lowering overhead. Interpretation is adopted to execute parts of the program that require symbolic execution, such as instructions that access to symbolic operands and to symbolic hardware. To achieve good performance in symbolic execution, it is indeed very important to optimize the performance of interpretation. S2E uses dynamic binary translation for different reasons [53] such as symbolic value creation, notify memory and register accesses, symbolic hardware and plugins notification.



```
addl   %edx,4(%eax)
movl   4(%eax),%edx
add    %eax,4
```

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwzx    r5,r2,r5      ;load operand from memory
lwz     r4,12(r1)     ;load %edx from register block
add     r5,r4,r5      ;perform add
stw     r5,12(r1)     ;put result into %edx
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwz     r4,12(r1)     ;load %edx from register block
stwx    r4,r2,r5      ;store %edx value into memory
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r4,r4,4       ;add immediate
stw     r4,0(r1)      ;place result back into %eax
addi    r3,r3,3       ;update PC (3 bytes)
```

Figure 3.3: Example of binary translation from x86 to PowerPC.

Figure 3.3 depicts an example of binary translation from a x86 source ISA to a PowerPC target ISA. The register values of the x86 architecture are maintained in a register context block stored in memory and then fetched into registers of the target PowerPC ISA. Frequently used registers guest registers are assigned to host machine registers. Other registers are stored directly in memory in the interpreter address space. In this example, r1 points to the first byte of the register control block, r2 points to the memory image of the source machine, and r3 contains the program counter. The mapping of source registers into target registers concretizes the *state mapping* between guest state and host state, that is necessary for virtualization. As shown in figure 3.3, the first instruction *lwz* loads the word starting at the first byte of the context block into *r*4, so the x86 register *eax* is mapped into *r*4. In the example, the program counter is contained into the PowerPC register *r*3 that is incremented at the end of each basic block.

In order to minimize the number of required translators, dynamic binary translators convert the source ISA code into an intermediate representation. This representation consists of a set of *micro-operations* that are operations simpler to emulate. Simpler instructions, such as *mov* and *ld* are translated into their equivalent counterpart. More complex instructions (e.g., instructions that are more complex to translate) are translated into call to *helper functions.* The latter emulates the behavior of the source ISA complex instruction.

From an architectural perspective, a dynamic binary translator consists of two main blocks: (i) the *dispatcher* and (ii) *code cache.* The dispatcher translates guest code into basic blocks. A *translation table* converts each guest instruction into the intermediate representation and then into the host ISA code. After the execution of each translation block the dispatcher regains control and selects the next block to execute.To reduce translation overhead, the dynamic binary translator contains also a *code cache.* The code cache implements fast-lookup of translated code, i.e, an hash table that maps program-counters into translated blocks.

Direct branch chaining allows the translated code to jump from one block to another within the code cache, improving the performance. To direct chain blocks, the dispatcher checks, before looking-up in the code cache, if the previous executed block performed a direct branch to the current translation block. Then, if the previous block jumps directly into the current block the dynamic binary translator modifies the end of the previous block with a jump to the current translation block.

QEMU

In this paragraph, we describe how QEMU [54] performs dynamic binary translation. S2E uses QEMU as a building block in its design. QEMU translates the guest code into an intermediate representation, named TCG (Tiny Code Generator) [55]. Guest code is translated into TCG and finally translated into the host architecture. A TCG "function” is a QEMU translation block. *Temporaries* are variables that only live within a translation block. Each function allocates explicitly temporaries. A *local temporary* is a variable that only lives in a function. On the other hand a *global* is a variable that is live in all the functions (equivalent of a C global variable). Globals are defined before function definitions. A TCG global can be a memory location (e.g. a QEMU CPU register), a fixed host register (e.g. the QEMU CPU state pointer). Instructions operate on temporaries, local temporaries or globals. In TCG, instructions are strongly typed and they can assume 32-bit value and 64-bit value. Pointers are alias to 32 bit or 64 bit integers. The pointer size is configured with the target word size. Instructions have a fixed number of output and input operands; except for the call instruction that can have a variable number of inputs and outputs. For instance, the instruction add_i32 $t0, t1, t2$ puts in $t0$ the sum of $t1$ and $t2$. TCG has the following instruction classes: (i) *register transfer operations* (ii) *arithmetic and logical operations*, (iii) *memory operations.* The first class consists of instructions used to transfer values from one register to others. The arithmetic and logical operation class contains all the instructions that perform math operations such as add, subtract, division, bitwise or, etc. Memory operations class allows to load and store values into the host and the guest memory. Further, QEMU performs constant folding and liveness analysis optimizations on TCG code in order to minimize the translation block size [56].

3.1.3 Symbolic Execution Engines

Interpretation of the bytecode allows to symbolically execute code. The interpreter fetches instructions from memory, decode them and then execute as mentioned in section 3.1. In this context, the memory is seen as an array of bytes that can be concrete or symbolic that are stored in data structures named *memory objects.* Memory is an array of memory objects.

Figure 1 depicts the interpreter loop for a symbolic execution interpreter. The interpreter runs in a loop that fetches instruction and extracts concrete or symbolic operands and from memory, and it executes operations such as arithmetic operations, logical operations, forking. As discussed in section 3.1 a symbolic execution engines must keep track of the *execution state* of each path, i.e., the registers state, memory and the path constraints. Execution states are used for updating registers as well memory locations on a per-path

Algorithm 1 Interpreter loop for a symbolic execution interpreter: reg and memory can contain both concrete and symbolic values, state represent the execution state captured by the symbolic execution engine

```
procedure InterpreterLoop($state, program\_counter$)
    while !$halt$ or !$interrupted$ do
        $opcode \leftarrow Fetch(program\_counter)$
        switch ($opcode$)
        case $load$:                    ▷ Load instruction: ld reg, mem_addr
        $(reg, mem\_addr) \leftarrow ExtractOperands(load, state, program\_counter)$
        $reg = state.memory[mem\_addr]$
        end case
        case $mov$:                     ▷ Mov instruction: mov reg0, reg1
        $(reg0, reg1) \leftarrow ExtractOperands(mov, state, program\_counter)$
        $state.regs[reg0] = reg1$
        end case
        case $store$:                   ▷ Store instruction: st reg, mem_addr
        $(reg, mem\_addr) \leftarrow ExtractOperands(store, state, program\_counter)$
        $state.memory[mem\_addr] = reg$
        end case
        case $arith\_op$:               ▷ Arithmetic instruction: arithm_op reg0, reg1, reg2
        $(reg0, reg1, reg2) \leftarrow ExtractOperands(arith\_op, state, program\_counter)$
        $expression \leftarrow PerformOperation(arith\_op, reg1, reg2)$     ▷ Expression can be
both a concrete value and a symbolic expression
        $state.regs[reg0] = expression$
        end case
        case $branch\_op$:              ▷ Branch with condition instruction: brcc label
        $destination \leftarrow ExtractDestination(program\_counter)$
        $program\_counter \leftarrow ForkAndSchedule(state)$
        end case
      ...
        end switch
        $updatePc(program\_counter)$
    end while
end procedure
```

basis, i.e., each state is modified independently to the others. Symbolic execution engines use copy-on-write mechanism to avoid redundant copies of the same memory objects, that significantly reduces overhead [30]. The procedure uses several helper functions that are explained later in this section. Algorithm 1 shows the symbolic interpretation routine. The variable *state* represent the current execution path selected by symbolic execution engine scheduler. The procedure *ExtractOperands* selects the operands according to the opcode passed as argument. The arguments are the operation type (e.g., *load*,*store*,*add*,*sub*, *bcc*, etc.), the execution state and the program counter that the procedure use to fetch operands from memory.

In the case of load/store instruction the symbolic interpreter read/write the value in the memory object located from/at the memory address mem_addr. Registers transfer operations simply copy concrete/symbolic data from a register to others. Arithmetic and logical operations build symbolic expressions. Symbolic expressions are concatenation of operations that are expressed in a format compatible for constraint solvers. When the interpreter executes an arithmetic or logic operation simply concatenates the left and the right operand of the operation and produces an expression that can be then transformed in format compatible for the constraint solver queries. The recursive generation of expression produces *expression tree* that encodes the expression. The procedure $PerformOperation$ executes an arithmetic or logic operations, it takes in input one or two expression trees (e.g.,unary or binary operations) and produces a new expression tree that encodes the result of the arithmetic or logic operation. An example are KLEE [26] expressions, when the symbolic execution interpreter executes $add\ reg1, 100$ with $reg1$ symbolic, KLEE produces an object that represents the expression that is encoded as $AddExpr(x, 100)$.

Branch instructions involve state forking. When the interpreter reaches a branch instruction it controls if the condition code is dependent on a symbolic variable. If positive, the branch instruction causes the call to the procedure $ForkAndSchedule(state)$ which does the following operations: (i) it computes the negated branch condition, (ii) it sends the combination of all constraints computed since that point to the constraint solver to see if it is feasible, (iii) if the constraint is feasible then the procedure forks a new state and selects which state execute next, otherwise it simply continues in the execution of the current state.

```
  mov eax, dword ptr 0xAB120890 ; load the value stored at AB120890 in eax
  add eax,100 ; eax = eax + 100
  je true_branch ; if eax == 0 then goto true_branch else continue
  mov ebx, 1  ; ebx = 1
  ret

true_branch:
  mov ebx, 2  ; ebx = 2
  ret
```

Figure 3.4: Symbolic execution of x86 code.

Figure 3.4 shows an example of Intel x86 assembler code that performs the following operations : (i) load the value stored at the address $0xAB120890$ in the eax register; (ii) add the value 100 to the value stored in the eax register and rewrites the eax register; (iii) if eax is equal zero jumps to the label *true_ branch* otherwise it continues the execution at the next instruction. Let suppose that at the address $0xAB120890$ is stored a symbolic value. The symbolic interpreter executes the mov instruction that loads the symbolic value stored at $0xAB120890$ in the eax register, that we denote with x. Then, the add instruction read x from eax and generates a symbolic expression $x + 100$. When the symbolic execution interpreter reaches the je (i.e., jump if equal zero) it forks two states: the first with the constraint $x+100 == 0$ and the second $x+100! = 0$. At this point the state scheduler decides which state must be executed first. In the case of depth-first search, the scheduler selects the path that allows to explore inner block, e.g., the path where the condition $x + 100 == 0$ holds. Then the control goes to the symbolic execution interpreter that executes the code for the scheduled path. The scheduler can preempt the symbolic interpreter and invoke the *InterpreterLoop* routine to execute code of a different path (i.e., to avoid that a path starves other paths). Interpretation is interrupted when virtual hardware triggers interrupt that is served by the dynamic binary translator.

3.2 The Enhanced Symbolic Execution Platform

As mentioned above, S2E analyzes translation blocks to decide whether a block can be executed natively or symbolically , i.e., when there is a read/write of a symbolic variable in a translation block. Translation blocks that access symbolic data need to be symbolically interpreted. Thus, the intermediate representation is translated into a bitcode that is interpreted as discussed in section 3.1.3. S2E uses QEMU for dynamic binary translation, LLVM as target bitcode and KLEE for symbolic interpretation as shown in figure 3.5.

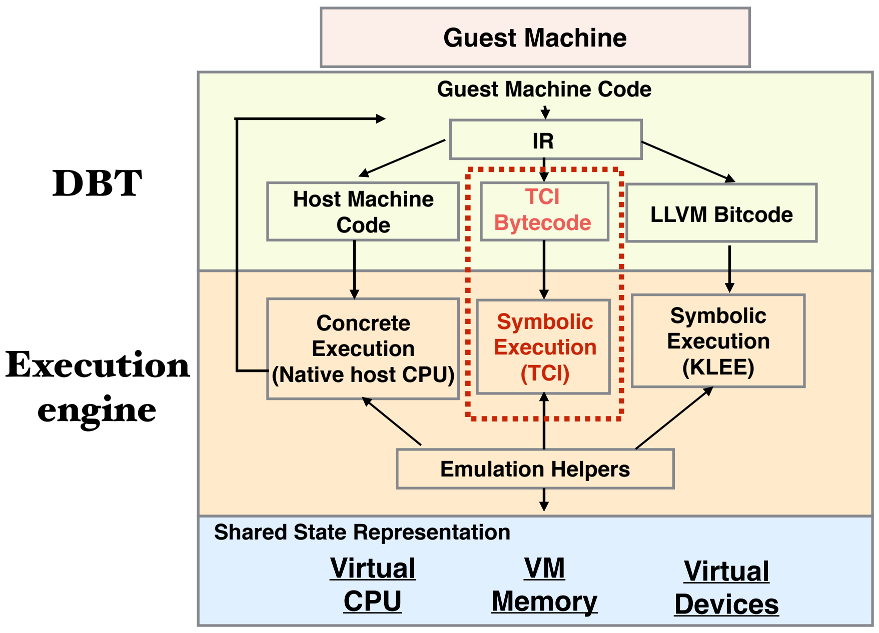


Figure 3.5: Enhanced platform design.

In order to leverage LLVM , S2E adds to QEMU a new translation layer. This layer translates the code from the QEMU intermediate representation (TCG) to LLVM. The choice of target representation has impact on translation performance. The total code execution time in symbolic mode is the sum of the translation time, the interpretation time and the constraint solving time. In this context, we are interested in minimizing the translation time from the guest code the the final host code when performing symbolic execution.

Research on virtualization spent time to optimizing dynamic binary translation process

[57]. However, only today research on symbolic execution face similar problems, and it was never subject of studies regarding how the translation can be made efficient.

```
0x000f0098: addr32 mov (%eax),%ebx
0x000f009c: add    $0x4,%ebx
0x000f00a0: cmp    $0x0,%ebx
0x000f00a4: jne    0xf00ad
```

Figure 3.6: Guest x86 code sample.

We need to take into account translation block caching effect that have impact on the overall performance measures. Now we describe an example of x86 to LLVM translation. The process consists of two translation steps: the first that translates x86 code to the TCG intermediate representation and the second that translates TCG code to LLVM code. The code in figure 3.6 performs an addition of a symbolic value λ contained in the register *eax* with the value 4. At the end of this operation the value contained in the register *eax* is the expression $\lambda + 4$. Figure 3.7 shows the translation of the code in the TCG intermediate representation. The code contains operations performed on x86 registers stored in memory. Besides the dynamic binary translator uses temporary registers to store intermediate results.

Finally, figure 3.8 shows the original x86 code translated into LLVM bitcode. The important thing to remark is that the code produced in each phase is equivalent to the initial guest code, which means that the code produce the same effects on a virtual machine with the same registers of the original guest machine.

Figure 3.5 shows the new platform based on S2E enhanced with the support for a new

```
00:  movi_i64 tmp5,$0x7f93ce2e3588
01:  st_i64 tmp5,env,$0x760
02:  mov_i32 tmp2,eax
03:  ld_i32 tmp4,env,$0x478
04:  add_i32 tmp2,tmp2,tmp4
05:  qemu_ld32 tmp0,tmp2,$0x0
06:  mov_i32 ebx,tmp0
07:  movi_i32 tmp1,$0x4
08:  mov_i32 tmp0,ebx
09:  add_i32 tmp0,tmp0,tmp1
10:  mov_i32 ebx,tmp0
11:  mov_i32 cc_src,tmp1
12:  mov_i32 cc_dst,tmp0
13:  movi_i32 tmp1,$0x0
14:  mov_i32 tmp0,ebx
15:  mov_i32 cc_src,tmp1
16:  sub_i32 cc_dst,tmp0,tmp1
17:  movi_i32 cc_op,$0x10
18:  movi_i32 tmp12,$0x0
19:  brcond_i32 cc_dst,tmp12,ne,$0x0
20:  movi_i32 tmp4,$0xa6
21:  st_i32 tmp4,env,$0x430
22:  goto_tb $0x0
23:  exit_tb $0x7f93962dfa48
24:  set_label $0x0
25:  movi_i32 tmp4,$0xad
26:  st_i32 tmp4,env,$0x430
27:  goto_tb $0x1
28:  exit_tb $0x7f93962dfa49
```

Figure 3.7: The code translated in the TCG code format.

bytecode which supports an instruction set that is ”semantically” closer to the intermediate representation instruction set. Figure 3.9 shows the translation process that translates the intermediate code in a new code format named TCI. LLVM uses SSA (static-single assignment), that is a form that arranges for every value computed by a program to have a unique assignment [56]. SSA is widely used in compilers because it simplifies a wide range of code optimizations. Conversely, TCI bytecode is very similar with respect to the intermediate representation.

The platform depicted in figure 3.5 reuses most of S2E components. We start modifying the dynamic binary translator to produce TCI bytecode. Afterwards, we plugged the

```
define i64 @tcg-llvm-tb-f0098-e-c9-f0000-44(i64*) {
entry:
%1 = getelementptr i64* %0, i32 0
%env_v = load i64* %1
%2 = add i64 %env_v, 1888
%3 = inttoptr i64 %2 to i64*
store i64 140272796054920, i64* %3
%4 = add i64 %env_v, 1072
%5 = inttoptr i64 %4 to i32*
store i32 152, i32* %5
%6 = add i64 %env_v, 0
%eax_ptr = inttoptr i64 %6 to i32*
%eax_v = load i32* %eax_ptr
%7 = add i64 %env_v, 1144
%8 = inttoptr i64 %7 to i32*
%tmp4_v = load i32* %8
%tmp2_v = add i32 %eax_v, %tmp4_v
%tmp0_v = call i32 @__ldl_mmu(i32 %tmp2_v, i32 0)
%9 = add i64 %env_v, 12
%ebx_ptr = inttoptr i64 %9 to i32*
store i32 %tmp0_v, i32* %ebx_ptr
%10 = add i64 %env_v, 1072
%11 = inttoptr i64 %10 to i32*
store i32 156, i32* %11
%tmp0_v1 = add i32 %tmp0_v, 4
store i32 %tmp0_v1, i32* %ebx_ptr
%12 = add i64 %env_v, 1072
%13 = inttoptr i64 %12 to i32*
store i32 160, i32* %13
%14 = add i64 %env_v, 32
%15 = inttoptr i64 %14 to i32*
store i32 8, i32* %15
%16 = add i64 %env_v, 36
%cc_src_ptr = inttoptr i64 %16 to i32*
store i32 0, i32* %cc_src_ptr
%17 = add i64 %env_v, 40
%cc_dst_ptr = inttoptr i64 %17 to i32*
store i32 %tmp0_v1, i32* %cc_dst_ptr
%18 = add i64 %env_v, 1072
%19 = inttoptr i64 %18 to i32*
store i32 164, i32* %19
%20 = add i64 %env_v, 32
%21 = inttoptr i64 %20 to i32*
store i32 16, i32* %21
%22 = add i64 %env_v, 32
%cc_op_ptr = inttoptr i64 %22 to i32*
store i32 16, i32* %cc_op_ptr
%23 = icmp ne i32 %tmp0_v1, 0
br i1 %23, label %label_0, label %24

; <label>:24                                   ; preds = %entry
%25 = getelementptr i64* %0, i32 0
%env_v2 = load i64* %25
%26 = add i64 %env_v2, 1072
%27 = inttoptr i64 %26 to i32*
store i32 166, i32* %27
store i8 0, i8* inttoptr (i64 27730344 to i8*)
ret i64 140271856515656

label_0:                                       ; preds = %entry
%28 = getelementptr i64* %0, i32 0
%env_v3 = load i64* %28
%29 = add i64 %env_v3, 1072
%30 = inttoptr i64 %29 to i32*
store i32 173, i32* %30
store i8 1, i8* inttoptr (i64 27730344 to i8*)
ret i64 140271856515657
}
```

Figure 3.8: The LLVM translated code for the code in figure 3.6.

```
addr32 mov (%eax),%eax
add    $0x4,%eax
je     0xf00aa
```

(a)

```
00:  movi_i64 tmp5,$0x7fac9ec5b588
01:  st_i64 tmp5,env,$0x760
02:  mov_i32 tmp2,eax
03:  ld_i32 tmp4,env,$0x478
04:  add_i32 tmp2,tmp2,tmp4
05:  qemu_ld32 tmp0,tmp2,$0x0
06:  mov_i32 eax,tmp0
07:  movi_i32 tmp1,$0x4
08:  mov_i32 tmp0,eax
09:  add_i32 tmp0,tmp0,tmp1
10:  mov_i32 eax,tmp0
11:  mov_i32 cc_src,tmp1
12:  mov_i32 cc_dst,tmp0
13:  movi_i32 cc_op,$0x8
14:  movi_i32 tmp12,$0x0
15:  brcond_i32 cc_dst,tmp12,eq,$0x0
16:  movi_i32 tmp4,$0xa2
17:  st_i32 tmp4,env,$0x430
18:  goto_tb $0x0
19:  exit_tb $0x7fac9ec5b588
20:  set_label $0x0
21:  movi_i32 tmp4,$0xaa
22:  st_i32 tmp4,env,$0x430
23:  goto_tb $0x1
24:  exit_tb $0x7fac9ec5b589
```

(b)

```
0x41add650: movi_i64 r0,
r136(0x7ff3a00bb588) st_i64 r0, r0 ,
0x760
0x41add665: call helper0
0x41add66e: ld_i32 r1 , 0x4780e
0x41add677: add_i32 r0, r0, r1
0x41add67d: qemu_ld32r0, (0x98), 0x0
0x41add68a: add_i32 r0, r0, r255
0x41add694: mov_i64 r1, r0
0x41add699: movi_i32 r2, r16(0x10)
0x41add6a1: st_i32 r2, r2 ,0x20
0x41add6aa: movi_i32 r2, r0(0x0)
0x41add6b2: st_i32 r2, r2 ,0x24
0x41add6bb: st_i32 r1, r1 ,0x28
0x41add6c4: st_i32 r0, r0 ,0xc
0x41add6cd: brcond_i32 to-> 0x41add702
if - r1 != r255
0x41add6df: movi_i32 r0, r166(0xa6)
0x41add6e7: call helper1
0x41add6f0: goto_tb: 0x0
0x41add6f7: exit_tb 0x7ff3a00bb588
0x41add702: movi_i32 r0, r173(0xad)
0x41add70a: call helper2
0x41add713: goto_tb: 0x0
0x41add71a: exit_tb 0x7ff3a00bb589
```

(c)

Figure 3.9: Translation x86 to TCI.

interpreter into the execution engine to symbolically execute code. We leverage the S2E abstraction of the execution state to store the program state and constraints. To implement forking functionalities we reused the existing procedures already available in S2E. To reduce implementation effort of building a new compiler for TCI code, we reused also LLVM emulation helpers.

3.3 Evaluation

This section explains the results obtained using the TCI bytecode representation. Section 3.3.1 describes results obtained with micro-benchmarks whereas the section 3.3.2 shows the results obtained for real Windows device drivers.

3.3.1 Micro-benchmarks

In this section we estimate the performance benefits obtained from the adoption of the TCI representation using two metrics:

1. **Concrete overhead:** that is the time spent in translating and interpreting only manipulating concrete data.

2. **Symbolic interpretation along a single path:** this metric measures the time spent executing the code with symbolic expression without switching states (e.g., following just one path).

To evaluate the concrete overhead metric we followed this procedure: (i) we created a bootloader that contains the assembly code of the micro benchmark, (ii) we activated the option that all the code executed must be interpreted by either the KLEE interpreter or the TCI interpreter.

```
void microbenchmark(){
        s2e_message("==================== Microbenchmark");
        int i = 2e5;
        int temp;
        int a = 4;
        int b = 5;
        while(i > 0){
                temp = a;
                a = b;
                b = temp;
                i--;
        }
        s2e_kill_state(0,"End of the microbenchmark");
}
```

Figure 3.10: A micro benchmark used for estimating the overhead in concrete mode.

Figure 3.10 shows a micro-benchmark that consists in swapping 200000 times two variables. This code stimulates both memory usage and caching effects.

	KLEE	**TCI**	**SPEEDUP**
Interpretation	317 sec	3,4 sec	100x
Translation	30 ms	95 us	310x

Table 3.1: Summary of performance enhancement in concrete execution.

As shown in table 3.1 the performance improvements with the new interpreter for the micro-benchmark depicted in figure 3.10 are around 100x for interpretation time and 310x for translation time.

To evaluate symbolic interpretation along a single path, we measured the time to open symbolic files of different sizes with forking disable both, to force the collection of constraints and to allow the overhead of just interpreting code.

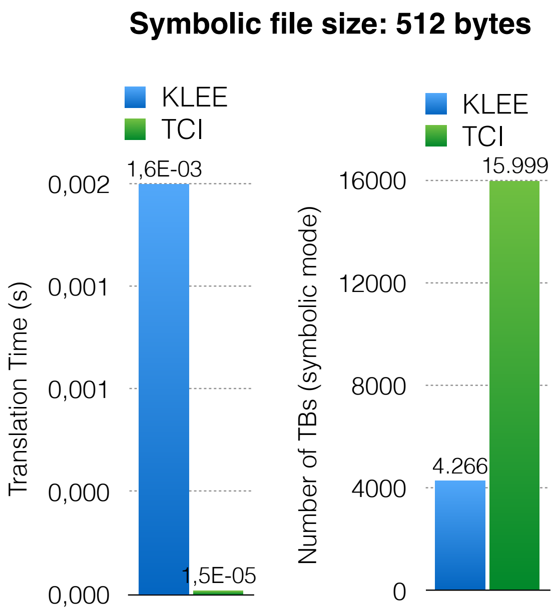


Figure 3.11: Opening a symbolic file of 512 bytes with forking disabled.

Figure 3.11 - 3.12 show the results about the time for opening of a 512/1024 bytes symbolic file. In both cases the translation time is several order of magnitude better. Furthermore, the number of translation blocks executed in symbolic mode are more because TCI translates more blocks that go into the symbolic execution interpreter. In all these experiments, forking is disabled because we want to force the execution just along a single path. When the interpreter reaches a fork point, the execution proceeds along a path and

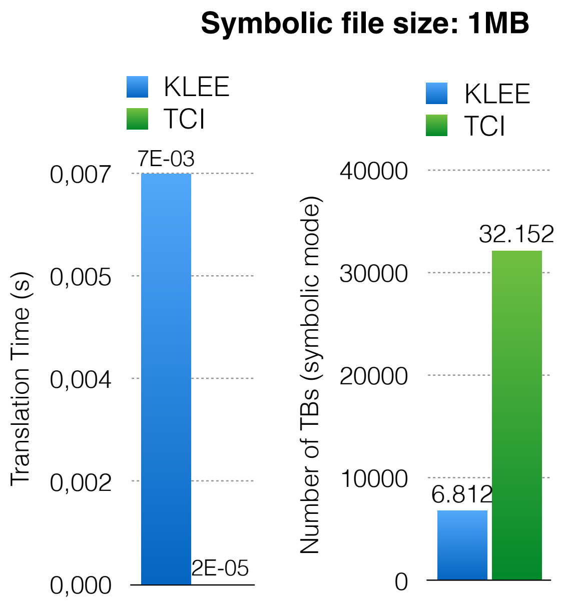


Figure 3.12: Opening a symbolic file of 1 megabyte with forking disabled.

the states are not forking. This configuration allows to evaluate how the interpreter performs with symbolic expressions, without involving the constraint solver.

3.3.2 Windows XP Device Drivers

This section shows the evaluation of three network Windows XP device drivers. The metrics that we choose for the analysis are the following:

1. **Speedup:** how many times TCI interpreter reduces the translation time.
2. **Completed paths:** the number of paths that terminate and that are killed by the symbolic execution engine.

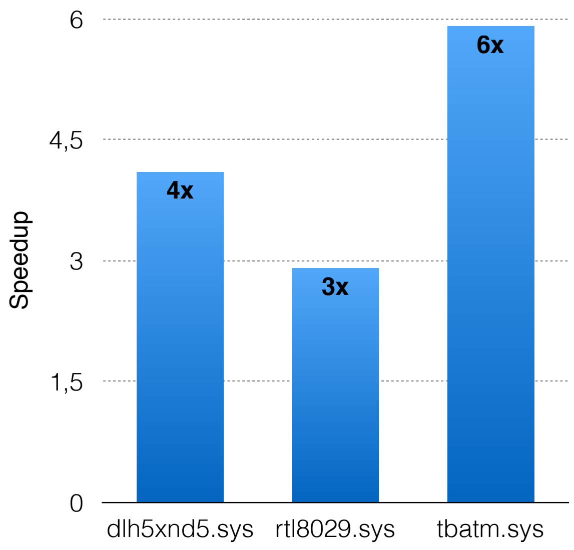


Figure 3.13: Speedup.

Figure 3.13 shows the ratio between the LLVM translation time and the TCI translation time. In these experiments, we compared the LLVM translation time with the TCI translation time. To leverage symbolic execution, we injected symbolic values at the device driver and OS interface (e.g., when a device driver entry point is called). We run the experiments for 120 minutes both for KLEE interpreter and for TCI interpreter. What emerges is that the average speedup is 3x with respect to the baseline.

Figure 3.14 depicts a comparative performance of the time spent in each phase of the symbolic execution process. The translation time is not took into account because is several order of magnitude less with respect to the time spent in code interpretation. In each case the interpretation time is less in the TCI case, that is, less time is spent in interpreting the code. Furthermore,TCI interpreter spends more time in constraint solving compared to

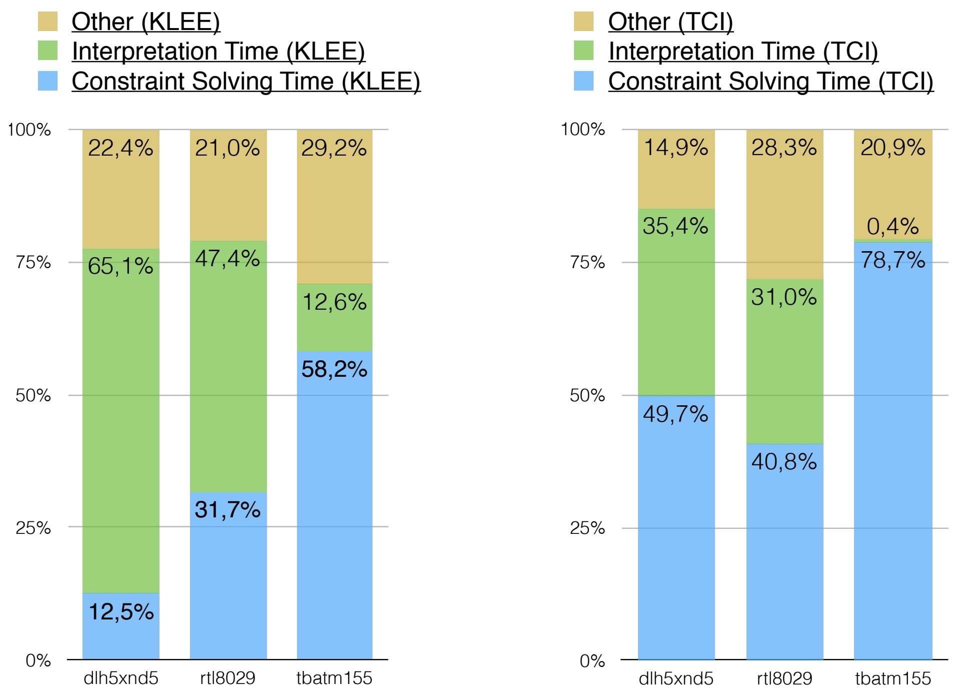


Figure 3.14: Breakdown of the time spent in each part of symbolic execution engine.

KLEE, that happens because more code is executed and then more queries are sent towards the constraint solver.

Figure 3.15 shows a comparative performance measure on the number of completed paths with KLEE and TCI. The experiments were performed using the same heuristics for both cases. For the drivers dlh5xnd5 and rtl8029, the number of completed paths with TCI interpreter is above 2x more than KLEE. The tbatm55 was a pathological case in which the state selection algorithm was unable to find a path that explores new code.

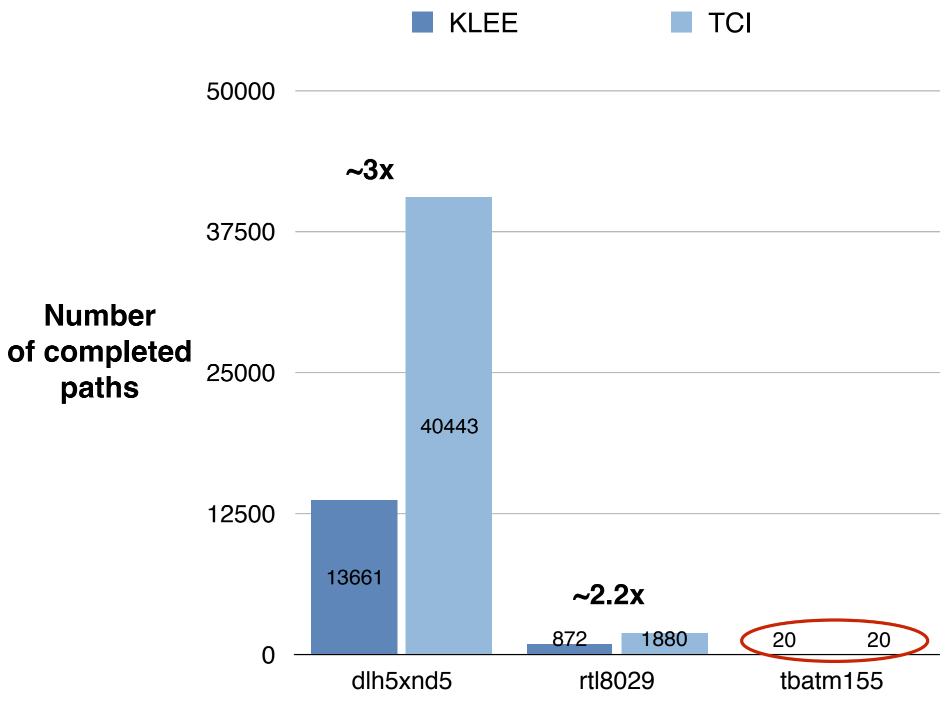


Figure 3.15: Comparative performance on completed paths .

Chapter 4

Model-Based Run-Time Verification of Device Drivers

This chapter describes a methodology that allows to automatically generate monitors for device drivers. Such monitors are used at runtime to detect anomalies in device drivers behavior. In section 4.1, we give a brief overview of the needed background. Then, section 4.2 provides a detailed description of the monitoring methodology. Finally, section 4.3 concludes this chapter with results obtained applying the methodology on a real class of device drivers.

4.1 Background

This section introduces main concepts required for understanding the details of the methodology. Section 4.1.1 describes runtime verification whereas section 4.1.2 gives a brief overview about specification mining.

4.1.1 Runtime Verification

Runtime verification is a system analysis and execution approach based on extracting information from a running system to detect and possibly react to observed behaviors when they satisfy or violate certain properties. Developers desire that their systems respect some particular properties such as datarace and deadlock freedom. Sometimes such properties are suitable to be best implemented algorithmically. Other properties can be more conveniently captured as formal specifications.

Runtime verification specifications are expressed in trace predicate formalisms, such as finite state machines, regular expressions, context-free patterns, linear temporal logics, etc., or extensions of these. Any mechanism for monitoring an executing system is considered runtime verification, including verifying against test oracles and reference implementations. When formal requirements specifications are provided, monitors are synthesized from them and infused within the system by means of instrumentation. Runtime verification can be used for many purposes, such as security or safety policy monitoring, debugging, testing, verification, validation, profiling, fault protection, recovery. Runtime verification is complementary with respect to the traditional formal verification techniques, such as model checking and theorem proving. Runtime verification analyzes only one or a few execution traces and then scales up giving more confidence in the results of the analysis at the expense of less coverage. A main advantage of runtime verification is that can be made an integral part of the target system, monitoring and guiding its execution during deployment.

4.1.2 Specification Mining

One of the most fascinating things in computing is the possibility of making the machine able to learn something such as automatically classifying objects such as text, voices, images, etc. The concept of *specification mining* [58, 59] is related to discovering formal specifications of the protocols that code must obey, from execution traces. Specification mining can be compared to learning the rules of English grammar by reading essays written by high school students; specification mining propose to focus on the essays of passing students and be skeptical of the essays of failing students. The specification miner observes program execution and builds a model (e.g., state machine, LTL logic predicate) that summarizes its behavior. These state machines are often called *behavioral models* and they describe the relationship between methods that change the state named ”mutators”, and methods that keep the state unchanged named ”inspectors”. Mining learns a finite-state automaton whose transitions are labeled with method names. Such an automaton approximates all legal functions call sequences and serves as a temporal specification. It must be noted that learned automata are not necessarily meant to be human-readable, as they are unlabeled [60] and are not guaranteed to be minimal [61].

4.2 The Proposed Methodology

This section describes the device driver monitoring process. The goal of the methodology is to automatically generate device drivers monitors from specifications mined on already existing device drivers. Such monitors can be useful to detect anomalies at runtime on device drivers e.g., to detect transient failures.

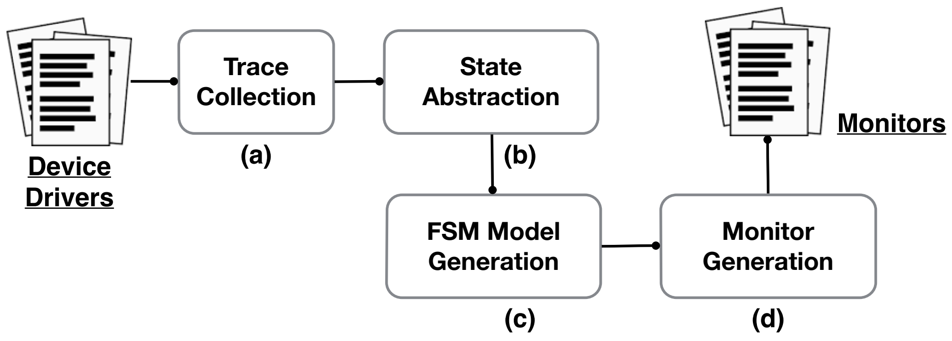


Figure 4.1: Workflow of the monitoring methodology.

Figure 4.1 depicts the workflow of the monitoring methodology. The process consists of four phases: (i) *trace collection*; (ii) *state abstraction*; (iii) *finite state machine model generation* ; (iv) *model generation.* In the trace collection phase, the system collects traces extracted from the execution of the device driver of a certain class such as network or disk. The state abstraction phase links the device-specific data, i.e., the state stored in device registers, to the abstract name as found on the specification standard. In other words, the raw execution data is converted in labeled data. The finite state machine model generation phase uses the extracted labeled execution trace to generate the finite state machine. Finally the monitor generation phase generates from the finite state machine a monitor that can be attached to a real device driver of the same class of the mined device driver.

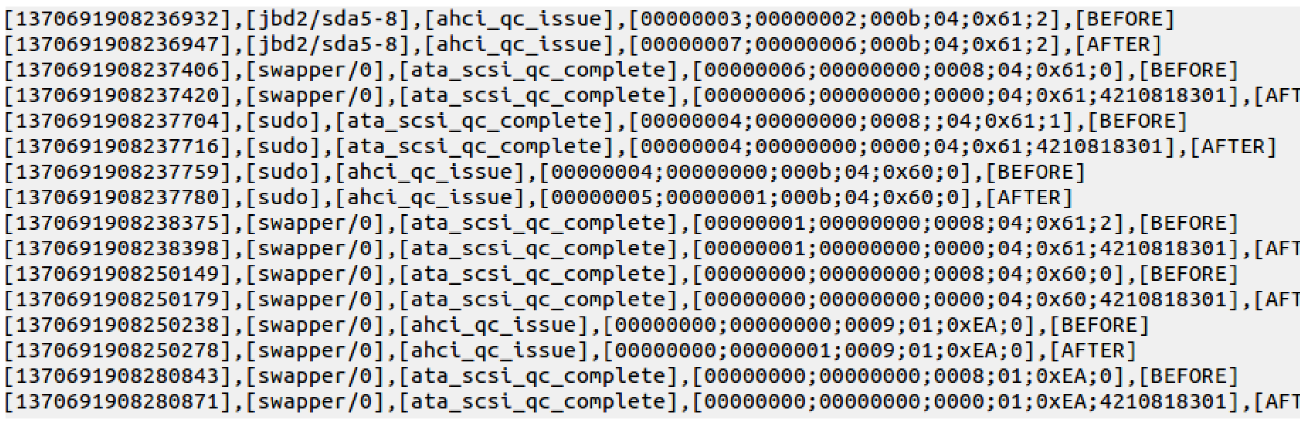
4.2.1 Trace Collection Phase

The trace collection phase gathers traces executing the device driver code. In order to collect traces, we place invocations to tracing functions in device drivers points of interest. The idea is to place tracing functions in points of the device drivers code that modify the state of the device registers. Each time the device driver code modifies the state of the device register, the event is logged into the execution trace.

Since device drivers maintain in their state also the device state, we look for function calls that modify the value of these variables. The format of a log entry is the following:

- *timestamp*: the time when the event happened.
- *process*: the process identifier of the process that is in execution.
- *function*: the name of the intercepted function.
- *state variables*: the value of the state variables.
- *event type*: this field allows to understand if an event happened before and after a certain call.

An example regards Linux SCSI driver [62, 63]. We selected as intercepted functions the *ahci_ qc_ issue*, *ahci_ qc_ complete*. These two functions modify the state of a SCSI port.



```
[1370691908236932],[jbd2/sda5-8],[ahci_qc_issue],[00000003;00000002;000b;04;0x61;2],[BEFORE]
[1370691908236947],[jbd2/sda5-8],[ahci_qc_issue],[00000007;00000006;000b;04;0x61;2],[AFTER]
[1370691908237406],[swapper/0],[ata_scsi_qc_complete],[00000006;00000000;0008;04;0x61;0],[BEFORE]
[1370691908237420],[swapper/0],[ata_scsi_qc_complete],[00000006;00000000;0000;04;0x61;4210818301],[AFT
[1370691908237704],[sudo],[ata_scsi_qc_complete],[00000004;00000000;0008;;04;0x61;1],[BEFORE]
[1370691908237716],[sudo],[ata_scsi_qc_complete],[00000004;00000000;0000;04;0x61;4210818301],[AFTER]
[1370691908237759],[sudo],[ahci_qc_issue],[00000004;00000000;000b;04;0x60;0],[BEFORE]
[1370691908237780],[sudo],[ahci_qc_issue],[00000005;00000001;000b;04;0x60;0],[AFTER]
[1370691908238375],[swapper/0],[ata_scsi_qc_complete],[00000001;00000000;0008;04;0x61;2],[BEFORE]
[1370691908238398],[swapper/0],[ata_scsi_qc_complete],[00000001;00000000;0000;04;0x61;4210818301],[AFT
[1370691908250149],[swapper/0],[ata_scsi_qc_complete],[00000000;00000000;0008;04;0x60;0],[BEFORE]
[1370691908250179],[swapper/0],[ata_scsi_qc_complete],[00000000;00000000;0000;04;0x60;4210818301],[AFT
[1370691908250238],[swapper/0],[ahci_qc_issue],[00000000;00000000;0009;01;0xEA;0],[BEFORE]
[1370691908250278],[swapper/0],[ahci_qc_issue],[00000000;00000001;0009;01;0xEA;0],[AFTER]
[1370691908280843],[swapper/0],[ata_scsi_qc_complete],[00000000;00000000;0008;01;0xEA;0],[BEFORE]
[1370691908280871],[swapper/0],[ata_scsi_qc_complete],[00000000;00000000;0000;01;0xEA;4210818301],[AFT
```

Figure 4.2: An example of Linux SCSI device driver execution trace

Figure 4.2 depicts an execution trace for the SCSI device driver. We choose as state variables the following: (i) PxSACT register value; (ii) PxCI register value; (iii) QC_FLAGS that is a bit mask of the command queue state; (iv) TASKFILE_FLAGS that is a bit mask of the task file state; (v) PROTOCOL that indicates the protocol used by the current command; (vi) COMMAND_TYPE that represents the SCSI command type; (vii) TAG a value to identify the command.

4.2.2 State Abstraction Layer

At this point execution traces contain only raw data, i.e., the binary value of each register and parameter. The abstraction layer is needed because several raw states can collapse in one state. The mapping of the raw data entries to the abstract state is many to one, i.e., several states represent a single state. We name this process' output file *abstract log file*. This process is inspired to the Adabu model mining algorithm [60]. The device driver model is built starting from the concrete state, that contains only numeric elements. Then, the state is abstracted using *abstraction functions* that allow to map each concrete state values to a label. The next phase of the process is to extract the transition function, that map one state into another.

<**source state**>,<**transition function**>,<**destination state**>

Figure 4.3: Abstract log entry.

At the end of this process the output trace file contains the entry depicted in figure 4.3.

Returning to the example of the Linux SCSI device driver, two registers PxSACT and PxCI are in charge of sending commands towards the disk. However, the transition function depends also on the TAG field. When the field is defined to a value different to the magic value ATA_TAG_POISON the state depends on which bit is high in the registers PxSACT and PxCI. If the PxSACT register bit in the TAG position is high, the value of the state is marked NCQ_COMMAND_ISSUED because the device drivers is sending an NCQ command. On the other hand, if the PxCI register bit in the TAG position is high, the value of the abstract state is marked NON_NCQ_COMMAND_ISSUED. Otherwise, the state is marked with a "don't care" value.

The field QC_FLAGS contains a bitmask that describes he state of the command queue. The flags associated to a command queue are depicted in figure 4.4. In the case of the SCSI

```
/* struct ata_queued_cmd flags */
ATA_QCFLAG_ACTIVE = (1 << 0), /* cmd not yet ack'd to scsi lyer */
ATA_QCFLAG_DMAMAP = (1 << 1), /* SG table is DMA mapped */
ATA_QCFLAG_IO = (1 << 3), /* standard IO command */
ATA_QCFLAG_RESULT_TF = (1 << 4), /* result TF requested */
ATA_QCFLAG_CLEAR_EXCL = (1 << 5), /* clear excl_link on completion */
ATA_QCFLAG_QUIET = (1 << 6), /* don't report device error */
ATA_QCFLAG_RETRY = (1 << 7), /* retry after failure */

ATA_QCFLAG_FAILED = (1 << 16), /* cmd failed and is owned by EH */
ATA_QCFLAG_SENSE_VALID = (1 << 17), /* sense data valid */
ATA_QCFLAG_EH_SCHEDULED = (1 << 18), /* EH scheduled (obsolete) */
```

Figure 4.4: Command queue flags.

device driver we are interested only in the ATA_QCFLAG_ACTIVE. This bit is high when the command is queue has not been notified yet to the SCSI subsystem, i.e., the command queue is still active. On the other hand, when the bit is low the command queue has been processed by the SCSI layer. The abstraction function maps three values:

1. *QC_SCSI_COMPLETE*: if the command has been both completed and notified to the SCSI subsystem the abstract state is called .

2. *QC_ACTIVE*: if the ATA_QCFLAG_ACTIVE is high.

3. *QC_NOT_ACTIVE*: if the command queue is considered neither active nor completed.

The *PROTOCOL* field contains a set of flags that determine how the data in transferred to the device , e.g., DMA, PIO. Finally the COMMAND_TYPE field contains the values of the possible commands that can be sent to the device that are compliant to the SCSI protocol.

4.2.3 Finite State Machine Generation

The abstract log file contains all the information needed to reconstruct the behavioral model of the interaction device driver/ device.

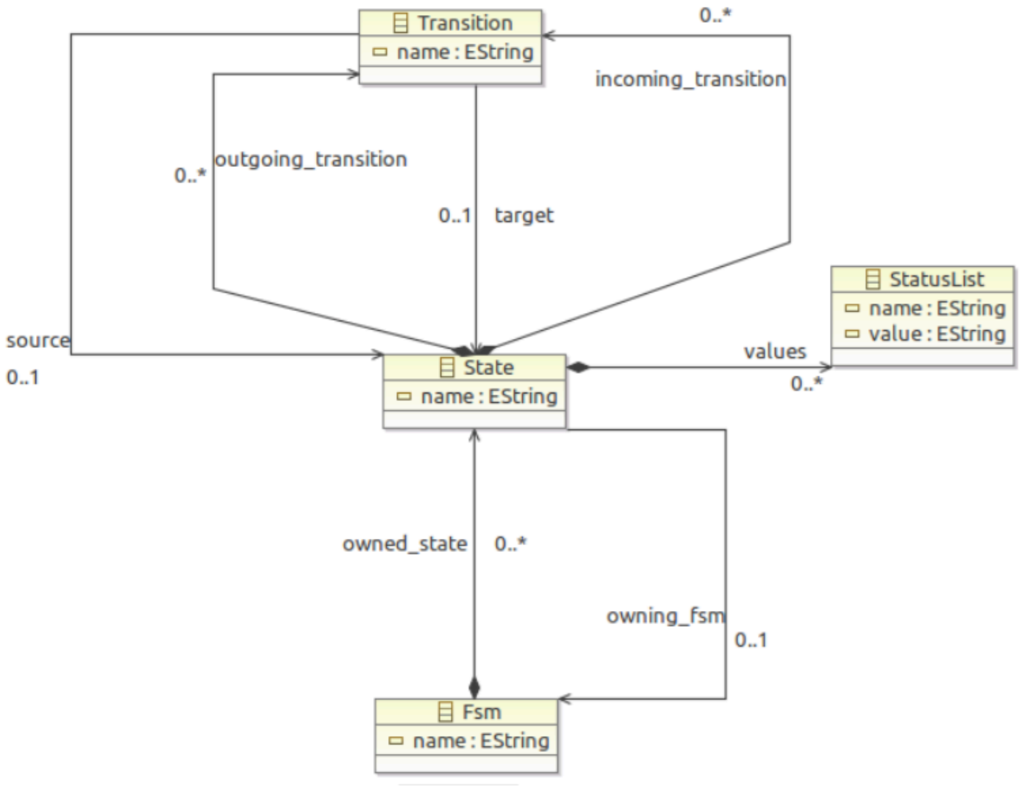


Figure 4.5: Metamodel of the finite state machines.

A metamodel that describes finite state machines is depicted in figure 4.6. The finite state machine generator uses the FSM metamodel to encode the state machine in a format suitable for the automatic code generation. we developed an algorithm that generates finite state machine from the abstract log file. The algorithm pseudocode is depicted in algorithm

2 . The procedure analyzes each log entry of the abstract log file. For each line the algorithm controls whether the source or the destination states have been already discovered. If so, it adds simply an entry of the outgoing/incoming transition for the source/destination state. If the state has not been created yet, the algorithm creates a new node with the name of the just discovered state.

Algorithm 2 State machine mining algorithm

```
1: procedure CreateFiniteStateMachine(abstract_log)
2:     FSM = createNewStateMachine()
3:     for each line in abstract_log do
4:         transition_name = getTransition(line)
5:         if transition_name ∉ Transitions then
6:             transition = createNewTransition(transition_name);
7:             Transitions.addTransition(transition)
8:         end if
9:         source_state_name = getSourceState(line)
10:        target_state_name = getTargetState(line)
11:        if source_state_name ∉ States then
12:            source_state = createNewTransition(source_state_name);
13:            States.addState(source_state)
14:        end if
15:        if target_state_name ∉ States then
16:            target_state = createNewTransition(target_state_name);
17:            States.addState(target_state)
18:        end if
19:        entry = [source_state, transition, target_state]
20:        if entry ∉ FSM then
21:            FSM.addEntry(entry)
22:        end if
23:    end for
24: end procedure
```

We applied the algorithm 2 to Linux disk SCSI device driver execution traces and we obtained the model depicted in figure 4.6.

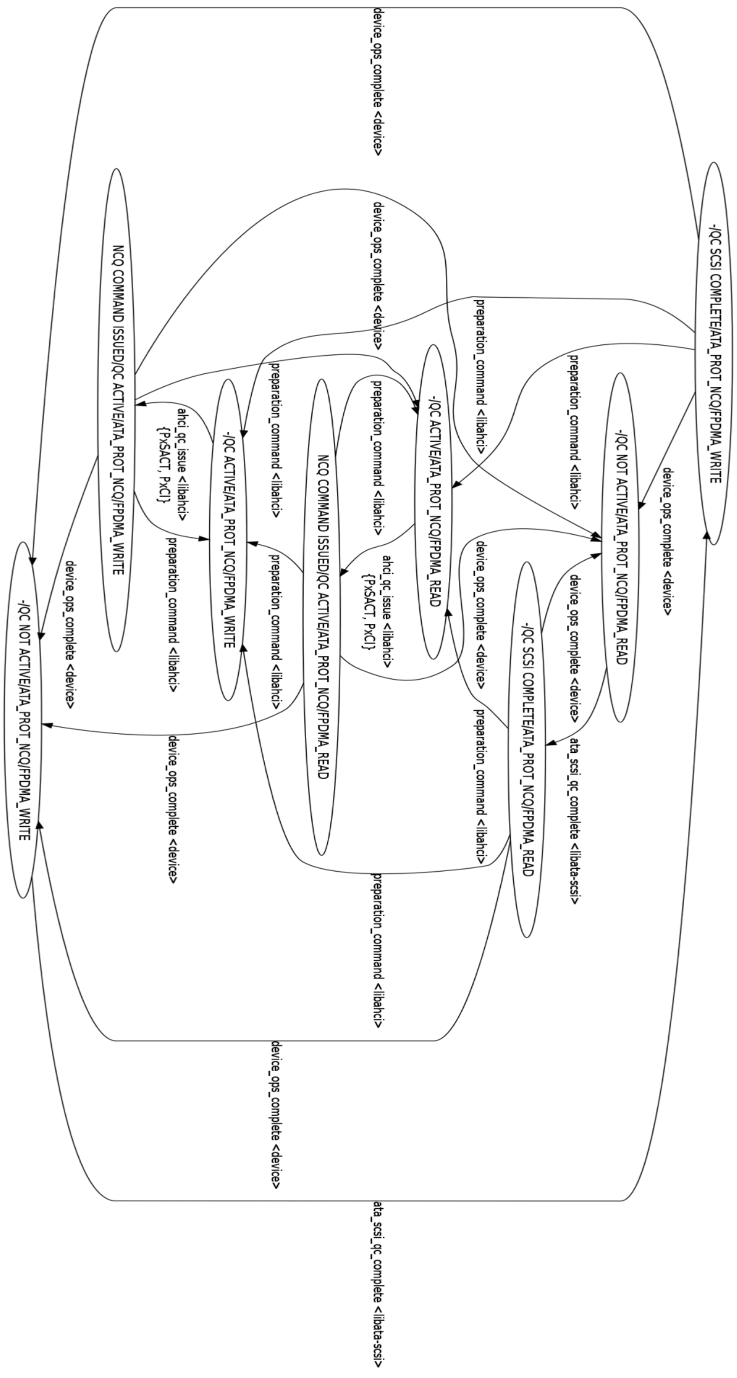

device_ops_complete <device>
NCQ COMMAND ISSUED/QC ACTIVE/ATA_PROT_NCQ/FPDMA_WRITE
device_ops_complete <device>
preparation_command <libahci>
-/QC SCSI COMPLETE/ATA_PROT_NCQ/FPDMA_WRITE
preparation_command <libahci>
preparation_command <libahci>
-/QC ACTIVE/ATA_PROT_NCQ/FPDMA_READ
preparation_command <libahci>
-/QC NOT ACTIVE/ATA_PROT_NCQ/FPDMA_READ
device_ops_complete <device>
ahci_qc_issue <libahci>
{PxSACT, PxCI}
-/QC ACTIVE/ATA_PROT_NCQ/FPDMA_WRITE
NCQ COMMAND ISSUED/QC ACTIVE/ATA_PROT_NCQ/FPDMA_READ
ahci_qc_issue <libahci>
{PxSACT, PxCI}
device_ops_complete <device>
device_ops_complete <device>
preparation_command <libahci>
preparation_command <libahci>
-/QC SCSI COMPLETE/ATA_PROT_NCQ/FPDMA_READ
device_ops_complete <device>
-/QC NOT ACTIVE/ATA_PROT_NCQ/FPDMA_WRITE
preparation_command <libahci>
ata_scsi_qc_complete <libata-scsi>
device_ops_complete <device>
preparation_command <libahci>
device_ops_complete <device>
ata_scsi_qc_complete <libata-scsi>

Figure 4.6: Model extracted from the Linux SCSI driver execution traces.

4.2.4 Monitor Generation Phase

The finite state machine generated in the previous phase describes the behavior that the device driver should manifest while communicating with a certain device type. The finite state machine is then translated into a monitor that checks whether the device driver behaves as expected at runtime. Whenever the monitor detects a transition that is not present in the model, it emits a warning message which informs the user that something strange happened. The model generator creates a piece of code that can be plugged into the kernel such as SystemTap [64] scripts or a kernel module. For each transition in the finite state machine the generator creates a *probe point* that is invoked before and after the call of the function. Before calling the transition function, the probe point function body selects, based on the current process and the current state, if this is a feasible transition. After the calling of the transition function, the probe point selects if given the current process id if the destination state is feasible. If the destination state does not belong to the model, the monitor triggers a warning.

4.3 Evaluation

This section describes the experiments performed to evaluate: (i) the overhead that the monitor introduces when the driver is running; (ii) a simple fault injection campaign to validate the monitor capability of detecting incorrect behavior.

4.3.1 Overhead

To evaluate the overhead introduced we emulated the concurrent database access, and then we measured the execution time of the sent query. We compared the average execution time for the sent queries when the monitor is enabled and when the monitor is disabled. The test configuration provides the following features:

- automatic generation of SQL queries.
- balanced load distributed between table insertion and deletion.
- growing the number of concurrent clients that access to the database.

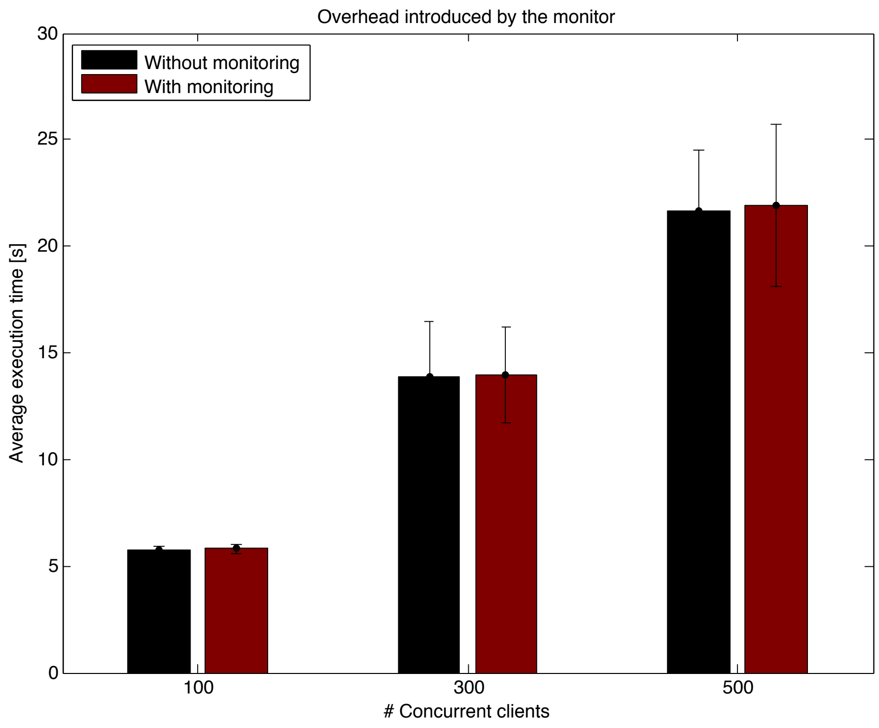


Figure 4.7: Overhead introduced by the monitor.

Concurrent clients number	h	p-value
100	0	0.5077
300	0	0.6299
500	0	0.3485

Table 4.1: Overhead of the monitor.

We repeated each test one hundred times. The results depicted in figure 4.7 show that the overhead introduced by the monitor is at most the 1.2%. We performed further statistical tests to estimate the similarities between the cumulative distributions of the average times. To evaluate the statistical equivalence, we applied the *T-test*. The null hypothesis is that the cumulative distribution of the difference between the distribution with monitoring and without monitoring is a Gaussian distribution with null expected value and unknown variance.

We performed the test with a level of significance that is 0.05. The table 4.1 shows that in every case the null hypothesis is rejected and therefore is valid that the distribution are equivalent with a confidence level of the 95%.

4.3.2 Effectiveness

This section evaluates the effectiveness of the driver by injecting faults inside the device driver. We performed our analysis on a virtual machine that emulates the behavior of the SCSI controller.

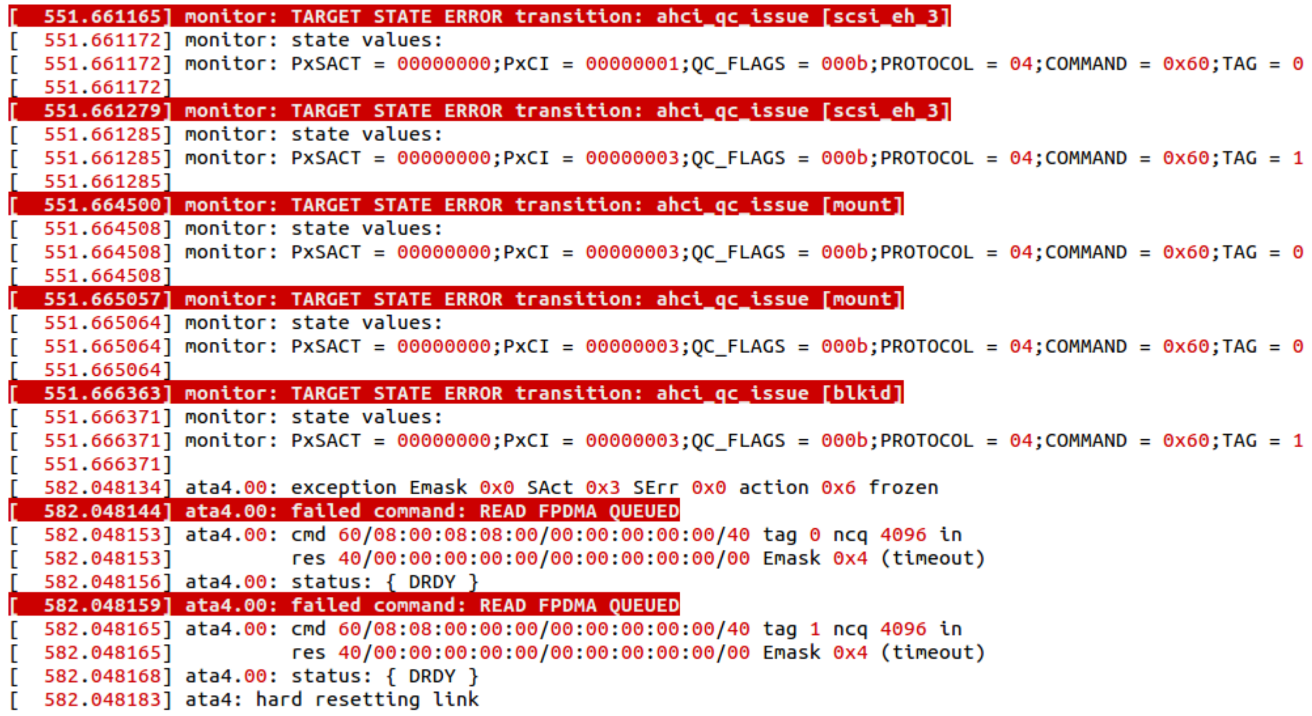
We injected a defect into the function *ahci_ qc_ issue* as shown in figure 4.8 . The defect

```

static unsigned int ahci_qc_issue(struct ata_queued_cmd ?qc) { .
...
if (qc?>tf.protocol == ATA_PROT_NCQ){
//writel(1 << qc?>tag, port_mmio +PORT_SCR_ACT);
writel (0 , port_mmio + PORT_SCR_ACT ) ;
}
return 0;
}
```

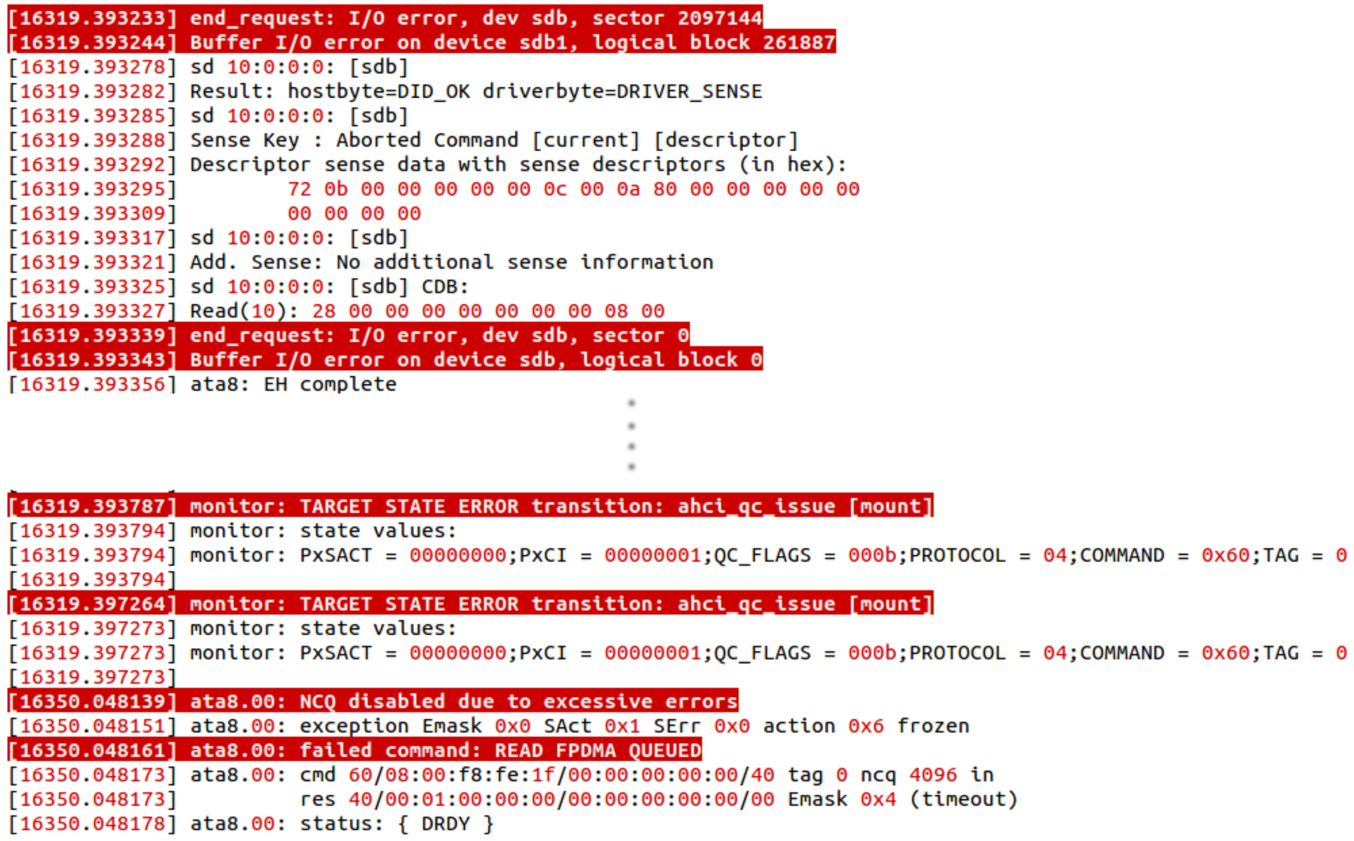
Figure 4.8: Command queue flags.

consisted in substituting the write of a register value from the correct value to zero. We configured our tests on virtual machine with two disks: (i) an IDE disk, (ii) a SATA disk where we perform the monitoring. The results of the tests depicted in figure 4.9 show that the monitor is able to identifies the wrong behavior, i.e. read FPDMA operations. Figure 4.10 shows a message log of the device driver error handing routine recognizes that there are continuous failures in the sending of NCQ commands. In this case the device driver simply refuses to send other NCQ commands and disables the NCQ mode. Then, the device driver continues to work using normal DMA setup as depicted in figure 4.11.



```
[   551.661165] monitor: TARGET STATE ERROR transition: ahci_qc_issue [scsi_eh_3]
[   551.661172] monitor: state values:
[   551.661172] monitor: PxSACT = 00000000;PxCI = 00000001;QC_FLAGS = 000b;PROTOCOL = 04;COMMAND = 0x60;TAG = 0
[   551.661172]
[   551.661279] monitor: TARGET STATE ERROR transition: ahci_qc_issue [scsi_eh_3]
[   551.661285] monitor: state values:
[   551.661285] monitor: PxSACT = 00000000;PxCI = 00000003;QC_FLAGS = 000b;PROTOCOL = 04;COMMAND = 0x60;TAG = 1
[   551.661285]
[   551.664500] monitor: TARGET STATE ERROR transition: ahci_qc_issue [mount]
[   551.664508] monitor: state values:
[   551.664508] monitor: PxSACT = 00000000;PxCI = 00000003;QC_FLAGS = 000b;PROTOCOL = 04;COMMAND = 0x60;TAG = 0
[   551.664508]
[   551.665057] monitor: TARGET STATE ERROR transition: ahci_qc_issue [mount]
[   551.665064] monitor: state values:
[   551.665064] monitor: PxSACT = 00000000;PxCI = 00000003;QC_FLAGS = 000b;PROTOCOL = 04;COMMAND = 0x60;TAG = 0
[   551.665064]
[   551.666363] monitor: TARGET STATE ERROR transition: ahci_qc_issue [blkid]
[   551.666371] monitor: state values:
[   551.666371] monitor: PxSACT = 00000000;PxCI = 00000003;QC_FLAGS = 000b;PROTOCOL = 04;COMMAND = 0x60;TAG = 1
[   551.666371]
[   582.048134] ata4.00: exception Emask 0x0 SAct 0x3 SErr 0x0 action 0x6 frozen
[   582.048144] ata4.00: failed command: READ FPDMA QUEUED
[   582.048153] ata4.00: cmd 60/08:00:08:08:00/00:00:00:00:00/40 tag 0 ncq 4096 in
[   582.048153]          res 40/00:00:00:00:00/00:00:00:00:00/00 Emask 0x4 (timeout)
[   582.048156] ata4.00: status: { DRDY }
[   582.048159] ata4.00: failed command: READ FPDMA QUEUED
[   582.048165] ata4.00: cmd 60/08:08:00:00:00/00:00:00:00:00/40 tag 1 ncq 4096 in
[   582.048165]          res 40/00:00:00:00:00/00:00:00:00:00/00 Emask 0x4 (timeout)
[   582.048168] ata4.00: status: { DRDY }
[   582.048183] ata4: hard resetting link
```

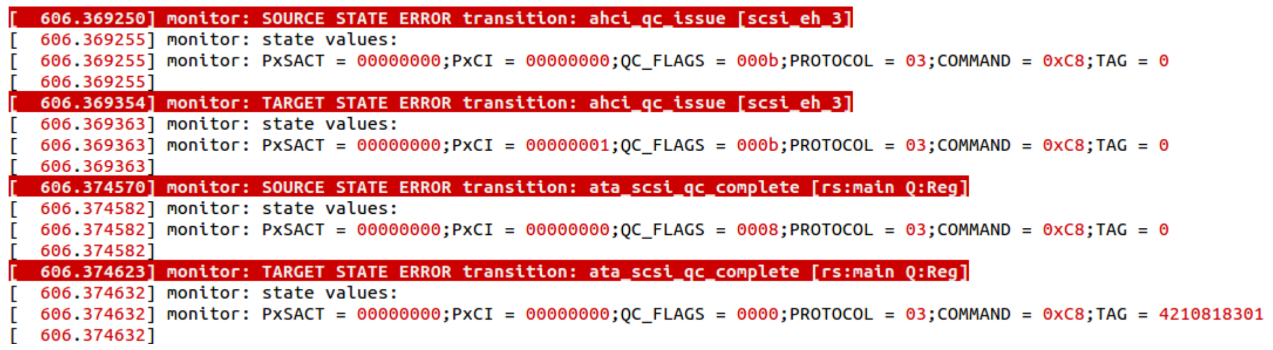
Figure 4.9: Monitor log part 1.



```
[16319.393233] end_request: I/O error, dev sdb, sector 2097144
[16319.393244] Buffer I/O error on device sdb1, logical block 261887
[16319.393278] sd 10:0:0:0: [sdb]
[16319.393282] Result: hostbyte=DID_OK driverbyte=DRIVER_SENSE
[16319.393285] sd 10:0:0:0: [sdb]
[16319.393288] Sense Key : Aborted Command [current] [descriptor]
[16319.393292] Descriptor sense data with sense descriptors (in hex):
[16319.393295]         72 0b 00 00 00 00 00 0c 00 0a 80 00 00 00 00 00
[16319.393309]         00 00 00 00
[16319.393317] sd 10:0:0:0: [sdb]
[16319.393321] Add. Sense: No additional sense information
[16319.393325] sd 10:0:0:0: [sdb] CDB:
[16319.393327] Read(10): 28 00 00 00 00 00 00 00 08 00
[16319.393339] end_request: I/O error, dev sdb, sector 0
[16319.393343] Buffer I/O error on device sdb, logical block 0
[16319.393356] ata8: EH complete
                                              .
                                              .
                                              .
                                              .

[16319.393787] monitor: TARGET STATE ERROR transition: ahci_qc_issue [mount]
[16319.393794] monitor: state values:
[16319.393794] monitor: PxSACT = 00000000;PxCI = 00000001;QC_FLAGS = 000b;PROTOCOL = 04;COMMAND = 0x60;TAG = 0
[16319.393794]
[16319.397264] monitor: TARGET STATE ERROR transition: ahci_qc_issue [mount]
[16319.397273] monitor: state values:
[16319.397273] monitor: PxSACT = 00000000;PxCI = 00000001;QC_FLAGS = 000b;PROTOCOL = 04;COMMAND = 0x60;TAG = 0
[16319.397273]
[16350.048139] ata8.00: NCQ disabled due to excessive errors
[16350.048151] ata8.00: exception Emask 0x0 SAct 0x1 SErr 0x0 action 0x6 frozen
[16350.048161] ata8.00: failed command: READ FPDMA QUEUED
[16350.048173] ata8.00: cmd 60/08:00:f8:fe:1f/00:00:00:00:00/40 tag 0 ncq 4096 in
[16350.048173]          res 40/00:01:00:00:00/00:00:00:00:00/00 Emask 0x4 (timeout)
[16350.048178] ata8.00: status: { DRDY }
```

Figure 4.10: Monitor log part 2.



```
[  606.369250] monitor: SOURCE STATE ERROR transition: ahci_qc_issue [scsi_eh_3]
[  606.369255] monitor: state values:
[  606.369255] monitor: PxSACT = 00000000;PxCI = 00000000;QC_FLAGS = 000b;PROTOCOL = 03;COMMAND = 0xC8;TAG = 0
[  606.369255]
[  606.369354] monitor: TARGET STATE ERROR transition: ahci_qc_issue [scsi_eh_3]
[  606.369363] monitor: state values:
[  606.369363] monitor: PxSACT = 00000000;PxCI = 00000001;QC_FLAGS = 000b;PROTOCOL = 03;COMMAND = 0xC8;TAG = 0
[  606.369363]
[  606.374570] monitor: SOURCE STATE ERROR transition: ata_scsi_qc_complete [rs:main Q:Reg]
[  606.374582] monitor: state values:
[  606.374582] monitor: PxSACT = 00000000;PxCI = 00000000;QC_FLAGS = 0008;PROTOCOL = 03;COMMAND = 0xC8;TAG = 0
[  606.374582]
[  606.374623] monitor: TARGET STATE ERROR transition: ata_scsi_qc_complete [rs:main Q:Reg]
[  606.374632] monitor: state values:
[  606.374632] monitor: PxSACT = 00000000;PxCI = 00000000;QC_FLAGS = 0000;PROTOCOL = 03;COMMAND = 0xC8;TAG = 4210818301
[  606.374632]
```

Figure 4.11: Monitor log part 3.

Chapter 5

Conclusions

This dissertation investigated the problem of improving operating system reliability focusing on techniques for detecting device drivers defects.

The dissertation first investigated a methodology to find device drivers defects based on the definition of the expected interaction between the device driver, the operating system kernel and the device. Symbolic execution has proven to be effective in the process of device driver defect detection, but the state-of-the-art techniques are not flexible enough to be adopted for detecting protocol bugs. The state of the art techniques require effort to be adapted for different kind of defects. For this reason, this thesis proposed a language for modeling expected interactions between device driver, operating system and device. The analysis was performed on a real device driver on two types of protocol bugs for the DMA API of the Linux kernel, namely DMA resource leaks and DMA race conditions.

Then, this thesis proposed an enhanced symbolic execution platform. This platform has a more efficient translation mechanism which reduces the gap between the code of the OS under verification (e.g., represented by x86 instructions), and the internal representation of the symbolic execution tool (e.g., using a custom ISA). The platform has been analyzed on several micro-benchmarks and on a set of three real device drivers. Results showed that in all the cases analyzed the enhanced platform that is on average 3x faster than the current state of the art.

Finally, this thesis described an approach to detect anomalies in device driver behavior. The approach extracts a model the expected behavior of the device driver under real workload. Then, the model is used to generate a monitor that inspects the state of the device driver in order to detect possible divergences from the expected behavior. The proposed approach was able to infer a model of the SCSI protocol implemented in the device driver. At the best of our knowledge nobody was able to verify with symbolic execution such a complex driver. This proves the complementarity of the approach with symbolic execution.

Bibliography

[1] Furhaad Shah. Google's self-driving cars expected in 2017, February 2015.

[2] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes*, 27(4):55–64, July 2002.

[3] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press, Redmond, WA, USA, 2004.

[4] Richard H. Cobb and Harlan D. Mills. Engineering software under statistical quality control. *IEEE Softw.*, 7(6):44–54, November 1990.

[5] International Standards Organization. *ISO 26262 Functional Safety Standard.*

[6] Radio Technical Commission for Aeronautics. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification.* 1992.

[7] NIST. Software errors cost u.s. economy 59.5 billion annually, October 2002.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[9] A. Mockus, N. Nagappan, and T.T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301, Oct 2009.

[10] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows xp kernel crash analysis. In *Proceedings of the 20th Conference on Large Installation System Administration*, LISA '06, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[11] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[12] 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 *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[13] Micheal Jones. What really happened on mars?, February 1997.

[14] Risat Mahmud Pathan. Mars pathfinder: Priority inversion problem, February 2015.

[15] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: testing drivers without devices. In *OSDI'12*.

[16] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *USENIXATC'10*.

[17] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 275–288, New York, NY, USA, 2009. ACM.

[18] 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. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[19] Asim Kadav and Michael M. Swift. Understanding modern device drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 87–98, New York, NY, USA, 2012. ACM.

[20] Linux Kernel Developers. Dynamic dma mapping using the generic device.

[21] LTP developers. Linux test project.

[22] T. Hanawa, H. Koizumi, T. Banzai, M. Sato, S. Miura, T. Ishii, and H. Takamizawa. Customizing virtual machine with fault injector by integrating with specc device model for a software testing environment d-cloud. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 47–54, Dec 2010.

[23] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[24] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 59–72, New York, NY, USA, 2009. ACM.

[25] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[26] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[27] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013.

[28] Suhabe Bugrara and Dawson Engler. Redundant state detection for dynamic symbolic execution. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 199–212, Berkeley, CA, USA, 2013. USENIX Association.

[29] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 239–254, New York, NY, USA, 2014. ACM.

[30] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1), February 2012.

[31] Radu Banabic, George Candea, and Rachid Guerraoui. Finding trojan message vulnerabilities in distributed systems. *SIGPLAN Not.*, 49(4):113–126, February 2014.

[32] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 101–114, Seattle, WA, April 2014. USENIX Association.

[33] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 14, February 2014.

[34] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.

[35] Martin Fowler. *Domain Specific Languages.* Addison-Wesley Professional, 1st edition, 2010.

[36] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[37] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend.* 2013.

[38] Christophe Henry. Boost meta state machine library, February 2015.

[39] Clang : a c language family frontend for llvm, February 2015.

[40] LLVM. Low-level virtual machine related publications, February 2015.

[41] B. Bartels and S. Glesner. Formal modeling and verification of low-level software programs. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 200–207, July 2010.

[42] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 115–128, New York, NY, USA, 2011. ACM.

[43] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[44] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38(1):53–64, March 2010.

[45] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 337–351, New York, NY, USA, 2011. ACM.

[46] Olatunji Ruwase, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry. Guardrail: A high fidelity approach to protecting hardware devices from buggy drivers. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 655–670, New York, NY, USA, 2014. ACM.

[47] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[48] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006.

[49] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 543–554, New York, NY, USA, 2012. ACM.

[50] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 147–162, Berlin, Heidelberg, 2008. Springer-Verlag.

[51] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 27–38, New York, NY, USA, 2011. ACM.

[52] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[53] Vitaly Chipounov. *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems.* PhD thesis, EPFL, 7 2014.

[54] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[55] Tiny code generator manual, February 2015.

[56] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[57] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, November 2012.

[58] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

[59] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[60] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 17–24, New York, NY, USA, 2006. ACM.

[61] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, June 1978.

[62] Serial ATA International Organization. *Serial ATA Revision 3.0.* `www.sata-io.org`.

[63] Intel Corporation. *Advanced Host Controller Interface for Serial ATA.* `http://www.intel.com/content/www/us/en/io/serial-ata/ahci.html`.

[64] SystemTap. Systemtap, February 2015.