Großer Beleg

Reverse Engineering of Device Drivers for the L4 Runtime Environment

Robert Muschner

October 25, 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Björn Döbel

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 25. Oktober 2012

Robert Muschner

Contents

List of Figures

1 Introduction

Operating systems need device drivers to support a wide range of available hardware. Due to the fact that drivers are most often developed for the Windows operating system other systems need to obtain this hardware support for themselves.

TUD:OS, the operating system developed at TU Dresden, addresses this problem by using a Device Driver Environment (DDE). DDE's API (application programming interface) wrappers make it possible to benefit from existing drivers from other operating systems by reusing the source code or the device driver's binary. Also, DDE provides an API to port device drivers directly or develop drivers from scratch.

APIs change over time. The target operating system's API change as well as the donor operating system's API. Hence, there is a remarkable ongoing maintenance effort for porting drivers. Especially maintaining the glue code between the target operating system and the donor operating system's driver.

In this thesis I'm going to apply RevNIC [CC10], a tool for analyzing and generating NIC (network interface controller) drivers. RevNIC generates C code that implements the original binary's hardware protocol without dependencies to the donor operating system. A device class-specific template provides the environment to embed the C code in the target operating system. The template does not consist of donor operating system's API redirections as contrasted with wrappers. Hence, the maintenance effort is reduced. The developer only maintaining a template for a specific device class which depends on the target operating system. The C code and the device class-specific template combined together result in a running device driver.

The aim of this thesis is to use RevNIC in practice and providing a network driver template for TUD:OS. Therefore, I generate a driver and port it to TUD:OS. Afterwards, I extract a template out of the driver. This TUD:OS template shall be usable with other network drivers generated by RevNIC. I evaluate the driver's performance and find out that the results are comparable with the Linux device driver. Finally, I summarize the thesis and give an outlook about future work.

2 Related Work

In this Chapter, I give an overview of how drivers are used on TUD:OS and which improvements are needed.

Writing drivers from scratch is no option because it requires a huge effort. For instance, the developer needs to obtain, read, and understand the hardware specification and the interfaces provided by the target operating system.

When porting device drivers the following properties should be achieved and provided by the device driver environment:

1. *Reuse* device drivers from other operating systems. The developer benefits from other's work and saves a lot of time. Also, he avoids to solve device driver-specific implementation problems.

2. The driver's *maintenance effort* should be as low as possible. Maintenance effort costs time and money that is usally not available.

3. The device drivers of TUD:OS should be easily *portable* to other operating systems. When a developer ports device drivers from TUD:OS to another operating system he should benefit from TUD:OS's device driver environment. This property guarantees to reduce the effort of porting device drivers from TUD:OS to another operating system.

The device driver environment provided by TUD:OS was first developed by Christian Helmuth. He implemented the *Device Driver Environment* (DDE) [Hel01] - an API wrapper for Linux 2.4. Thomas Friebel developed a DDE for FreeBSD [Fri06] and extracted DDEKit. Further investigation by Bernhard Poess showed [Poe07] that it is also possible to run device driver binaries on top of a device driver environment by using a lightweight virtual machine. Implementation details and properties in reference to the properties mentioned above of these approaches are described in Section 2.1.

The maintenance of these DDEs is still a remarkable effort because they must be adjusted to any change of the donor and target operating systems' APIs. This problem is addressed by RevNIC [CC10] and I will describe its details in the 2nd part of this chapter.

2.1 DDE

An operating system can interact with many different devices and provides the user with their functionality. The communication between operating system and device is managed by a device driver. You can use the device's functionality with the help of the

appropriate driver. Most of the time, drivers are already developed for major operating systems, such as Windows, but not implemented for the operating system of your choice. The Device Driver Environment approach addresses this problem by providing a platform-independent layer for device drivers.

DDE's idea is to wrap existing drivers. In Christian Helmuth's diploma thesis [Hel01], this approach is implemented in *dde-linux2.4*. He developed libraries which redirect API calls from the donor operating system to DROPS (Dresden Real-Time Operating System), the former name of TUD:OS. The source code is also needed, because TUD:OS' DDEs provide modified header files and reuse code from Linux. This approach fullfills property 1 but is restriced to Linux 2.4.

Thomas Friebel improved the approach by developing a DDE for FreeBSD [Fri06]. He noticed many similarities between DDE for Linux 2.4 and DDE for FreeBSD. Any driver needs to use threads, memory, interrupts and I/O resources. This knowledge leads to one of Thomas Friebels results, the separation of DDE into two parts. One is *DDEKit* (host-specific) and the other is *dde_**. The latter is a pattern for the donor operating system-specific wrapper. Once a dde_* is implemented you can port any driver from the donor operating system which relies on the implemented API. DDEKit is a construction kit for device drivers which implements the interface between the operating system and the ported driver. Therefore, device drivers developed for DDEKit are portable to other operating systems whereon DDEKit is available which fullfills property 3.

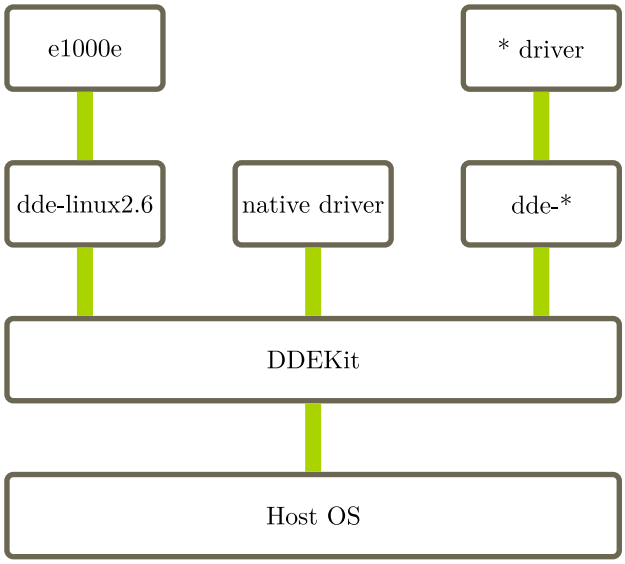

Figure 2.1: DDE Framework

As shown in Figure 2.1, DDEKit is the foundation which provides an interface between the target operating system and the device driver. Each dde_* is implemented using DDEKit's API. Consequently, every dde_* is portable to any operating system whereon DDEKit is available. As Figure 2.1 indicates, different drivers can run in parallel and you can implement a new driver on top of DDEKit which remains host-independent.

On the downside, the maintenance costs remain if you want to provide an interface for the newest device drivers. On every change in the donor operting system's API the developer has to adjust the wrapper. Also, the driver's source code is required as well as in the approach of Christian Helmuth but "Binary Device Driver Reuse" [Poe07] overcomes this issue. Bernhard Poess uses a different device driver environment (DD/Env) which consists of a virtualized device driver, a lightweight virtual machine monitor (VMM) and the OS personality emulation [Poe07]. DD/Env runs as a normal application in user-mode and offers the driver's functionality as a server to clients. For instance, Linux modules are embedded into a driver layer which loads and executes the binary whereby all undefined symbols are resolved and exported. The OS personality emulates the Linux kernel to the driver and the VMM scans the binary's instructions and replaces them with emulation code. This approach allowed reusing closed source drivers and should be adaptable by the DDE's mentioned before. Therefore, the number of reusable drivers is increased instead of only using Linux 2.4 device drivers.

2.2 RevNIC

The DDE framework covers the properties 1 and 3 from the introduction of this chapter. Properity 2 is not covered by DDE, therefore remarkable maintenance effort still exists.

RevNIC [CC10] uses automated analysis and a code generator to reduce the drivers' maintenance effort. The driver of interest is analyzed in a virtual machine and its functionality is mapped to a synthetic driver specified for the target operating system. RevNIC operates on the binary level to analyze the device driver's code.

The driver is analyzed once and can be ported onto every operating system using a connection layer - a *template*. The developer uses the generated donor-independent driver and hooks it into the host OS using a class-specific template. Hence, the maintenance effort is reduced to adopting the template. RevNIC is a prototype implementation and only supports analyzing network drivers.

2.2.1 Symbolic Execution

As King described in his paper [Kin76], symbolic execution was developed for program testing. By testing a program, the absence of bugs can be shown, but it does not prove its correctness. On the other hand, formal analysis can prove a program's correctness but it is impractical because the analyst needs to provide a formal proof on an abstract mathematical model of the system which is a nontrivial task. Symbolic execution is a practical approach between these two. A program is executed symbolically instead of testing with a set of inputs. The outcome is an execution tree which consist of all reachable paths and their appropriate inputs. Each possible input leads to a test case. The analyst can validate these tests against its expectations for correctness either formally or informally.

Symbolic execution's main idea is to trigger any possible path in the tested program. Instead of concrete input such as '9' or '"a"', symbolic execution uses *symbolic values* to encode all possible data. As result the tester obtains knowledge about each single path and its input dependencies. A specific range of inputs can be assigned to each path. This

range is computed out of symbolic values by a constraint solver. The number of possible values decrease by all operations and conditional branches the symbolic variables pass through. During the analysis *dead code*, paths which will never reached on execution, can be detected.

Below, I will show how programs are executed symbolically by considering the code in Figure 2.2.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc,char* argv[])
4  {
5    int i = atoi(argv[1]);
6    i = i * 10;
7
8    if( i < 100 )
9    {
10     i = i * 10;
11   }
12   else
13   {
14     i = i - 10;
15   }
16
17   printf("%d",i);
18   return 0;
19 }
```

Figure 2.2: Example code

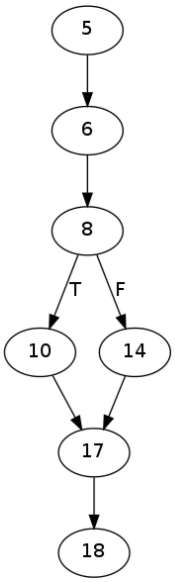

Figure 2.3: Execution Tree

The tester invokes a symbolic execution engine which executes the program with symbolic values for *argc* and *argv*. In line 5 the local variable i is defined and declared to $i = \lambda$ because our symbolic input is assigned to it, called λ in this example. The atoi() function and any code behind is analyzed symbolically as well but for simplicity we just work with its result. On line 6, i is modified and changed to $i = \lambda * 10$.

On each conditional statement (e.g. *if*) symbolic execution is forked, in the case of *if* into two states. One state is created for the true branch and one for the other. The definition of 'state' is up to the symbolic execution engine but it contains at least all symbolic variables.

The constraint solver evaluates the *if* statement in line 8 and thereupon the execution engine generates 2 states. When the condition's true branch is analyzed i becomes $i = \lambda * 10 < 100$ and $i = \lambda * 10 \geq 100$ in the case of the else branch. Therefore, a constraint solver is needed to see which paths can be executed.

In line 16, the program is in one of two states. Each contains variant symbolic variables: $i = (\lambda * 10 < 100) * 10$ if the condition is true and $i = (\lambda * 10 \geq 100) - 10$ for the else branch. In the end of our main function, printf() is called - another library function. Symbolic execution analyzes it such as atoi() in line 5 and generates a huge amount of states.

For each symbolically executed program you can generate an *execution tree*. Nodes are executed statements labeled with their statement number or line number. Each *if* condition is a node with two branches. One labeled with 'T' (true) and the other with 'F' (false). An execution tree for our example program is shown in Figure 2.3. For each generated leaf there exists at least one concrete value which terminates the program in this state.

2.2.2 KLEE

The authors of "Symbolic Execution and Program Testing" [Kin76] implemented a symbolic execution engine, called EFFIGY, which symbolically executed programs written in a simple version of IBM's PL/1. A modern symbolic execution engine is KLEE [CDE08]. It analyzes programs compiled to LLVM bitcode, the intermediate code representation of the LLVM compiler infrastructure. "The LLVM Project is a collection of modular and reusable compiler and tool chain technologies" [Lat12]. The project's core library offers code optimization and code generation functionalities which are used by RevNIC's code generator as well.

KLEE generates states for the code which should be analyzed. These states contain a register file, stack, heap, program counter, and path condition. RevNIC's authors modified KLEE that instead of delivering the hole program RevNIC delivers basic blocks one by one.

KLEE takes the bitcode, number and type of symbolic arguments. When the engine needs to convert symbolic values into concrete ones, the STP [GD07] constraint solver is invoked. This happens on conditional statements and at the end of the test to provide the tester with sample input triggering each analyzed path.

If the analysis reaches a branch point KLEE forks the currently executed state and needs to decide which state is explored next. During this part of the analysis KLEE faces the problem to repeatedly choose the same state. For instance, if the branch point is an infinite loop, the analysis could get stuck in this state. Therefore, KLEE uses two *heuristics* in a round robin fashion to not get caught in one part of the program. *Random Path Selection* maintains states in a binary tree and selects the next state by traversing the tree from the root and randomly choosing the path to follow at branch points. At each branch point both subtrees have equal probability of being selected. This strategy favors states placed closely to the root, therefore the chance to cover unexplored code is higher and the execution will not get stuck in infinite loops. *Coverage Optimized Search* assigns the probability to cover new code to each state. It randomly selects states according to their value.

While exploring the code KLEE checks for dangerous operations (e.g. dereferenced null pointer) and notifies the tester if there is a concrete input that leads to a program crash. As result, KLEE provides test cases for all covered paths and replays them on demand.

2.2.3 Selective Symbolic Execution

As described in Section 2.2.1, symbolic execution evaluates all possible paths through the program. Library dependencies are analyzed as well. Executing symbolic execution costs time, but in most cases a tester would like to examine either the program or the library. For instance, if you want to analyze the example from Figure 2.2 in-depth knowledge about printf() is not necessary to understand the program. Therefore, the standard library does not need to be analyzed and accordingly the analysis' execution time is reduced. The benefit is analyzing programs faster and just obtain results for the important part. Due to this fact, a better approach is to execute the program symbolically and the libraries concretely.

RevNIC exactly implements this observation. It executes the driver in its native environment but the driver is selectively analyzed. In other words, the driver's code is executed symbolically, but any call to its environment is executed with concrete values and without the respective analysis' overhead.

The execution engine needs to switch between code located in the *symbolic domain* (the driver) and the *concrete domain* (the environment). Total control over the execution is required to intercept and switch between these two execution modes. A virtual machine is more suitable for this purpose than a native environment.

To implement the switch between symbolic and concrete domain, RevNIC needs to transform concrete values into symbolic ones and the other way around.

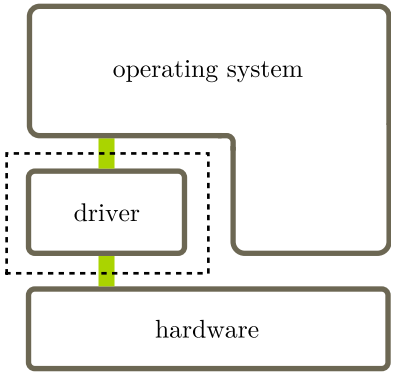

Figure 2.4: Symbolic and concrete domain

There are two interfaces between symbolic and concrete domain as shown in Figure 2.4: the connection between the driver and the operating system and the connection between the driver and the hardware.

A driver's call to the donor operating system invokes a constraint solver to choose feasible concrete values out of the symbolic arguments. If the call to the donor operating system returns, the concrete values are forced to be symbolic. When any error state is reached, RevNIC terminates the execution path and proceeds with a different state.

In the second interface the driver communicates with the hardware. It would be impractical to obtain the corresponding device for each driver you want to analyze. Drivers analyzed by RevNIC communicate with so called *symbolic hardware* and they do not require real hardware. RevNIC intercepts hardware requests from the driver and

returns symbolic values to make sure every response is considered. A DMA request from the driver results in symbolic values as well if RevNIC is aware of the donor operating system's DMA API.

2.2.4 Code Synthesizer

The driver analysis described in the previous section results in a LLVM bitcode file and data about the execution tree. RevNIC includes a code generator which produces a C-code file that implements the driver's functionality covered by the analysis.

The generated code is not directly able to work. For example, unmapped macros are used which needs to be adjusted. They operate as placeholders for system-specific function calls because RevNIC does not know anything about the target platform. To address this problem, the developer has to provide a template which wraps the generated driver with system-dependent code. The outcome is a *synthetic driver*. Examples for successful ports can be found in the paper [CC10].

2.2.5 The Big Picture

In this Section I will explain how RevNIC's components explained in the previous sections fit together.

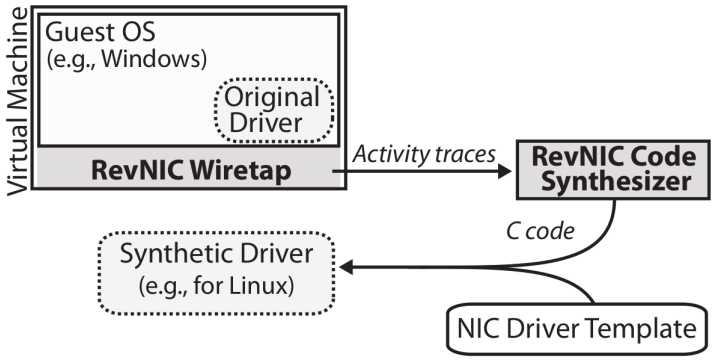

Figure 2.5: RevNIC's architecture [CC10]

As Figure 2.5 shows, the donor operating system with the driver of interest, called guest operating system in this figure, is installed in a virtual machine. The wiretap layer contains a modified KLEE and the domain switch functionality as described in Section 2.2.3. Also, RevNIC applies a modified version of Qemu [Bel05] in this layer.

Qemu executes code from the concrete domain in a normal manner with concrete values provided by RevNIC. On the other side, when the donor operating system calls the driver (from concrete to symbolic domain) then the driver's *entry points* are invoked. RevNIC is aware of these entry points by manually inspecting the respective driver class' API. Entry points are addresses of the network driver's functions such as initialization routine or send/receive functions. Qemu redirects the currently executing driver code to a dynamic binary translator (DBT) which transforms the binary code into LLVM

bitcode. This way, it can be analyzed by KLEE. The modified KLEE symbolically executes basic blocks and returns with an updated state.

RevNIC uses *heuristics* to decide which state to execute subsequently. "RevNIC's heuristics aim to choose the paths most likely to lead to previously unexplored code" [CC10, Chapter 3.2]. The first heuristic selects paths most likely to discover new code. Parts of the code can be analyzed more than once for example loops or a part where symbolic values needed to be concretized. Therefore, RevNIC executes every possible path first and thereafter already executed paths. The separation heuristic chooses paths that step out of loops and discards next iteration's paths. Another heuristic triggers interrupts to exercise the driver's interrupt handlers. The point in time when to insert interrupts depends on the driver's type. The RevNIC authors reported that triggering after the send routine works well in the case of network drivers. A final heuristic skips unnecessary functions such as debug/log routines.

Other often called hardware-related functions can be replaced by models to speed up execution. For instance, I/O port operations are often called network driver-specific functions. RevNIC can provide the developer with statistics of frequently called functions which could be replaced by such models.

After the analysis, the bitcode file and log files are processed by RevNIC's code synthesizer. A C code file is generated which is wrapped in the template and results in a synthetic driver.

2.2.6 Method of Operation

In this section I will describe the work flow by example: generating C code out of the Windows RTL8139 network device driver.

At first I prepared a virtual disk with an installed Windows. During analysis, only the driver and its communication with its environment are important. Anything else such as the Windows Help and Support service or the user account control extend analysis time. Hence, I disabled these unnecessary services to speed up the analysis. Another improvement is to start the analysis with a snapshot of the system to avoid boot time. The snapshot should be an already booted Windows with a prepared environment to invoke the driver. For example, we want to invoke the send routine, a driver's entry point, to start the analysis. Therefore, I saved a snapshot with an already opened command line window and a typed ping command. Now, the analysis could be repeated with different parameters by running this snapshot.

As described in Section 2.2.3, no real hardware is required. However, we need Windows to load and initialize the RTL8139.sys module. Hence, we provide RevNIC's Qemu with a fake device, a description of the network device such as the vendor ID. RevNIC makes Windows believe that such a device is attached. Windows in turn loads the proper module.

The analysis results in LLVM bitcode and log files about the execution. The driver's functionality is available without any specific API dependencies but not usable yet. As a last step, we need to run the code synthesizer which reconstructs the execution tree and generates C code. The generated driver code is not applicable. Next, we need to provide a *template*.

2.2.7 Template

A template is a target system-dependent wrapper. The developer has to implement the connection between the generated placeholders and the system functions. For instance, functions that read and write to I/O ports needs to be connected to the host system-specific functions. Also, resources - such as memory and interrupts - need to be allocated. Furthermore, the developer needs to provide driver class-specific functions. For example, network drivers require a handler for incoming packets which hands a packet over to higher-level layers for processing.

The analysis itself is done in one hour but the most time consuming part is wrapping the template around the generated code. As described in the [CC10], one developer needed 1 week to port the RTL8139 Windows driver to Linux.

3 DDEKit template for network device drivers

In this chapter I present the practical work of this thesis. At first, I did simple experiments to get in touch with the development environment. I used two drivers from the RevNIC paper [CC10], which the authors made available to me. The NE2K and the RTL8139 Windows drivers had both been ported to Linux already. Once I gained experience in porting these drivers to TUD:OS in two different ways, I used RevNIC to obtain my own reverse engineered driver. This process failed because of technical reasons I will describe in Section 3.3. We decided to skip the analysis step and continued with a bitcode file provided by the RevNIC authors. I was then able to generate a working driver for DDEKit. From this generated driver I extracted a driver template - a connection between DDEKit and a RevNIC-generated network device driver.

3.1 RTL8139 on top of dde-linux2.6

TUD:OS contains a network multiplexer called *Ankh*. Ankh provides multiple virtual network cards and makes use of DDE to run Linux network drivers. I used Ankh to test the Linux RTL8139 network driver generated by RevNIC.

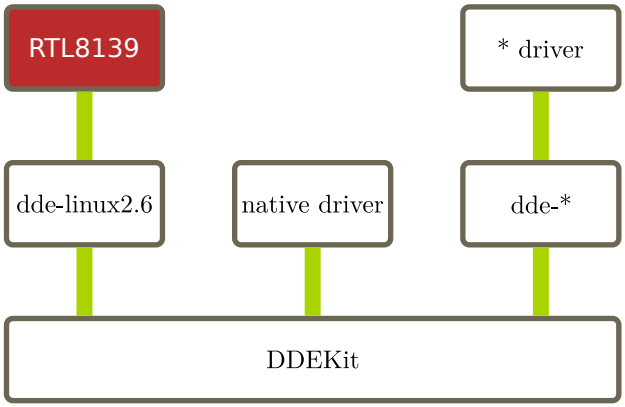

Figure 3.1: RTL8139 on dde-linux2.6

Figure 3.1 shows the driver's environment within TUD:OS. To test the driver's correctness, I executed the ping-pong application an example program provided by Ankh. Two clients, ping and pong, infinitely send empty packets to each other.

I started two instances of Qemu. Each instance was provided with the network interface controller (NIC) model RTL8139 and a TCP connection between them. Ankh

was compiled with the newly generated driver for this device. The output of Wireshark [Fou12] showed successfully received packets on each instance which indicates a working driver.

3.2 NE2K on top of DDEKit

The second step was to directly execute a driver on DDEKit without any donor operating system-specific wrapper. For that purpose, I used the NE2K driver. The authors had ported this Windows driver onto Linux and had evaluated its performance against the original Linux driver. The RevNIC paper reports nearly the same throughput but RevNIC's driver had performed slightly better.

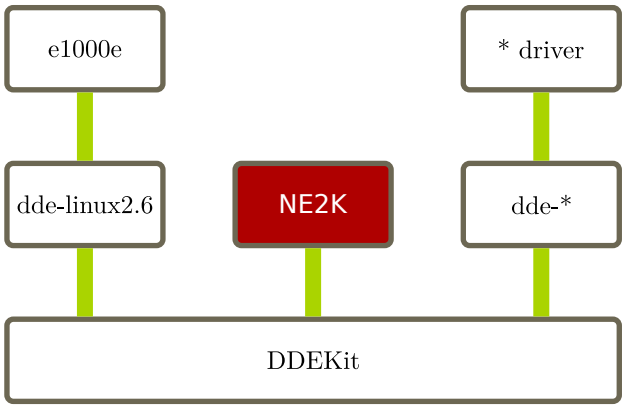

Figure 3.2: Experiment 2

As can be seen from Figure 3.2, the driver should work on DDEKit without any wrappers. Because this driver is generated by RevNIC the code is split in the driver's functionality and a Linux template. Hence, I needed to replace the Linux template with a DDEKit template. This template connects the NE2K driver with DDEKit.

I manually analyzed the driver for its entry points such as sending/receiving a packet, the initialization routine and interrupt handling. The template forwards hardware interrupts to the driver code and also delivers network packets to the TCP/IP stack and to the driver. Therefore, the template needs to call these functions. I found the methods listed in Table 3.1.

Entry point	Description
Ne2kOSInit	The driver's initalization routine
NdisEntryMiniportHandleInterrupt	The dirver's interrupt handler
Ne2kSend	The driver's send method
Ne2kReceive	The driver's receive function
NetReceivePacket	non-implemented packet handler routine

Table 3.1: The driver's entry points

The method *Ne2kReceive* contained a non-implemented call to *NetReceivePacket*. This routine is implemented by the Linux template and handles incoming packets. Consequently, I implemented the function by myself. It forwards received packets to the test application.

The template is also responsible for allocating and providing access to device resources, such as interrupts. Therefore, I allocated memory for the device driver structure, attached the interrupt, and made the driver aware of the device's I/O base address by querying the PCI configuration registers. The memory allocation and interrupt functions are shown in code Listing 3.6. As result, the driver directly runs on DDEKit.

The test application, *Arping* [Hab12], sends ICMP packets and expects a response before sending a new one. A server side implementation had already been available within DDE. The test environment was set up with an instance of Qemu executing TUD:OS with the driver and Arping's server inside. The server waits for incoming packets and replies to arping which runs on the host.

I used Wireshark [Fou12] to validate that packets were successfully delivered. The properly working network driver consists of the generic NE2K driver code and the DDEKit template.

3.3 Generating C code with RevNIC

In the first two experiments I obtained knowledge about porting a Linux driver onto DDEKit and proved myself that RevNIC-generated drivers function correctly. The next step was to generate a synthetic driver with RevNIC and thereby make it generally possible to port any network device driver from any operating system to DDE.

While compiling RevNIC, many warnings and errors occurred, especially from LLVM-2.4. I used the recommended *gcc* and *g++* versions but the errors persisted. In particular, most errors came from *libstdc++*. Instead of restructuring my host system I decided to use a virtual machine. Inside, I ran Ubuntu 9.04 where all dependencies to compile RevNIC were fulfilled.

With a compiled RevNIC I setup its environment as described in Section 2.2.6. I installed Windows XP into a virtual machine where the driver I wanted to port comes from. To recap the setup, I run Ubuntu 11.10 on my host, Ubuntu 9.04 in a virtual machine and Windows XP in a nested virtual machine.

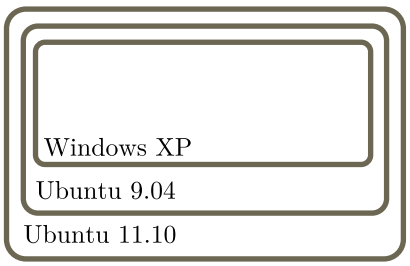

Figure 3.3: Test environment

The first run successfully finished and I obtained a *module.bc* file, the LLVM bitcode which represented the analyzed part of the driver. The next logical step was to generate C code out of module.bc using RevNIC's code synthesizer. When I executed the analyzer I got a segmentation fault. I tested the analyzer by generating code from the module.bc the authors of RevNIC made me available. The result was a successful execution and a module.c file. Hence, there was something wrong within the analysis phase. I realized the problem to be a lack of code coverage because the log files indicated a small number of basic blocks compared to the numbers reported in the RevNIC paper. I obtained 498 basic blocks as contrasted with 1065. Further investigation showed that the analysis didn't cover the driver's send routine. Consequently, the module.bc obtained during my analysis setup would not work as a network driver.

In further attempts I ran into segmentation faults or memory out of bound errors. Rarely, I obtained a module.bc but all of them substantially lacked basic blocks. The overall number was about 500 most of the time. In one execution the result was ca. 700 basic blocks, however that still wasn't enough to work on. The different results come from RevNIC's use of heuristics as described in 2.2.5.

During my experiments, several problems occurred with the virtual machines. Sometimes, analysis stopped due to the lack of memory space. RevNIC produced log files of about 5GB during the analysis but as the number of basic blocks indicate the memory consumption will be much higher with full coverage. Another quirk is Qemu itself. It tend to corrupt the windows image every other execution. The authors advise to backup an already booted windows snapshot and to start each analysis with a copy of this backup. However, those steps required additional effort.

While I tinkered with RevNIC, I was aware of S^2E [CKC12]. This project generalizes RevNIC's approach and is developed by the same authors at EPFL, Switzerland. Therefore, I intended to analyze the driver with S^2E and to produce the C code with RevNIC's code synthesizer. S^2E was successfully compiled on Ubuntu 11.10 but it turned out that the log files produced by S^2E are incompatible with RevNIC's code generator.

I spent a lot of time running RevNIC with different configurations. For instance, I tried different combinations of Windows, RevNIC and Qemu settings. All these approaches didn't work and we began to debug RevNIC. We found out that there was something wrong with the state management. Formerly discovered and saved states couldn't be found and RevNIC terminated after a given timeout. After a week of debugging we decided to skip the analysis and work on with an already generated bitcode file. It would take more effort to dive into this problem than permitted in the time frame of this thesis.

In summary, we were not able to get the 2009 version of RevNIC up and running within a reasonable amount of time.

3.4 Generating a DDEKit driver out of LLVM bitcode

After no bitcode file could be generated by RevNIC, I used the module.bc of the author's experiments. This file was generated from a RTL8139 Windows network driver. In this section I describe how I generated a DDEKit-based driver from this bitcode.

I used RevNIC's code analyzer to transform the LLVM bitcode file into C code. The result is one file with generated C code. This file contains 82 functions all named such as *function_** where the star indicates the function address within the original driver's binary. A snippet of this code can be seen in Figure 3.4.

```
1  ...
2  /* Function has 1 parameters */
3  /* Function returns 4 bytes */
4  uint32_t function_10322(uint32_t Param0)
5  {
6   DECLARE_VARS(0x14);
7
8     /* Predecessors:  */
9     /* LBL0x10322: */
10    //PC=0xfc81c322
11    env->t2 = (env->esp + 0xfffffffc);
12    env->esp = env->t2;
13    //PC=0xfc81c323
14    env->esi = Param0;
15    //PC=0xfc81c327
16    env->eax = *(uint32_t*)((env->esi + 0x10));
17    ...
```

Figure 3.4: Generated code snippet

Each function's return value is either *uint32_t* or *void* and the number of parameters varies from 1 to 9 whereby each parameter is typed with *uint32_t*. To comprehend where each generated statement comes from, the corresponding address is added as a comment. There are also macros used for local variable's declarations and system-specific calls to I/O ports. The last thing to mention are method calls to the Windows API (NDIS*) which need to be replaced with the target operating system's functionality. All together, the generated C code consists of about 8612 SLOC (Source Lines of Code) measured by SLOCCount [Whe12].

Due to the fact that the target operating system is TUD:OS, I used the following functions as replacement.

Macro	Replacement
WRITE_PORT_UCHAR(p, v)	l4util_out8(v, p)
WRITE_PORT_USHORT(p, v)	l4util_out16(v, p)
WRITE_PORT_ULONG(p, v)	l4util_out32(v, p)
READ_PORT_UCHAR(p)	l4util_in8(p)
READ_PORT_USHORT(p)	l4util_in16(p)
READ_PORT_ULONG(p)	l4util_in32(p)
DbgPrint_RTL8139(...)	ddekit_log(DEBUG_RTL8139,__VA_ARGS__)

Table 3.2: Macros replaced by Host OS functionality

While I investigated the code I noticed comment lines without corresponding code statements. Therefore, I looked up the addresses in the binary's assembler code and it turned out there was missing functionality. It seemed that the experiments' code coverage achieved by RevNIC's authors did not reach 100%. At this point in time I reached the situation which is described as the most time consuming part of the manual template instantiation in RevNIC's documentation. I benefited from the RTL8139 Linux driver which was one of the drivers the original RevNIC authors had generated. Hence, the missing lines of code were completed with less effort by copying them into the C code generated by me.

To make the code more readable, I replaced function names with their corresponding symbols. The tool PE Explorer [Sof12] helped me to extract all symbols. Windows drivers are usually compiled with debugging symbols which are not provided within the binary. The symbols can be retrieved from Microsoft®'s symbol server [Cor12].

In the second experiment described in Section 3.2, I ported the NE2K Linux driver to DDEKit. Therefore, I already had code which implements resource allocation and a send/receive interface to the Arping test application. There was only one resource missing which I needed to implement additionally - DMA (direct memory access). The way to use DMA on TUD:OS is to allocate continuous and pinned memory (data space). This data space needs to be attached to the corresponding region manager and queried for its physical and virtual address. The function calls I used are shown in the code Listing 3.6 line 13, 15, and 17. For further information about data spaces and region manager have a look at [APJ+01].

I replaced the Windows API calls with driver functions or wrappers that forward to TUD:OS methods.

Windows API function	Replacement
NdisAllocateSharedMemory	AllocateSharedMemory
NdisReadPciSlotInformation	RTL8139ReadPciInformation
NdisWritePciSlotInformation	RTL8139WritePciInformation
NdisMDeregisterInterrupt	RTFast_DisableInterrupt

Table 3.3: Windows API replacements

The first three replacements are wrappers implemented by myself. *AllocateSharedMemory* writes the physical and virtual address of a DMA region to pointers. These pointers are parameters provided by the caller of the function. *RTL8139ReadPciInformation* and *RTL8139WritePciInformation* implement calls to *ddekit_pci_* read* and *write*. The last replacement is a driver function that disables interrupts coming from the device.

As described before, Arping was used to validate the driver's basic functionality.

3.5 NIC template for DDEKit driver generated by RevNIC

To support automated generation of different drivers using RevNIC, we need a host-OS specific template as described in Section 2.2.7. As shown in Figure 3.5, the template's purpose is to glue the API used by the driver to the host OS.

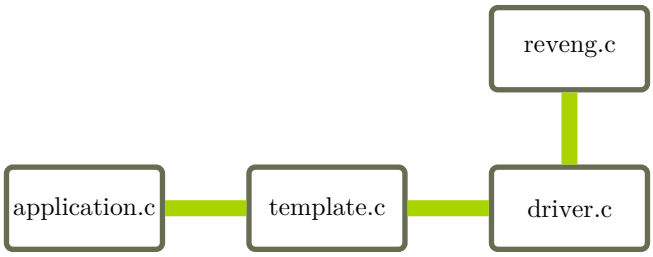

Figure 3.5: Template's relation

The driver's reverse engineered functionality is implemented in *driver.c*. The file *reveng.c* and *reveng.h* are used for operating system-specific function calls and debugging. The header file also implements the macros shown in Table 3.2. These macros were used by RevNIC's code synthesizer to replace calls to the donor operating system's API.

The template consists of 6 functions as shown in Table 3.4.

Function	Description
rtl8139_init	initalizes the driver and its environment
init_dev	setup resources and calls the drivers initialization routine
close_dev	frees resources
net_tx	calls the drivers send routine
net_rx_handle	disables interrupts and invokes the drivers interrupt handler
NetReceivePacket	forwards network packets to the application

Table 3.4: Basic network driver template functions

The template provides a program entry point, the *main()* function in code Listing 3.6, which initializes DDEKit and calls the initialization function. The initialization function performs the following steps:

1. Allocate datastructures
2. Initialize device lock
3. Allocate memory for DMA
4. Query PCI configuration space to obtain the port's I/O address and size
5. Request this I/O address for use
6. Call hardware initialization routine (function provided by the driver)

7. Attach callback function to IRQ (receive routine)
8. Enable interrupt

Figure 3.6 shows important parts of the template I implemented.

```
[...]
static struct net_device *dev;

int init_dev()
{
  int err;

  /* allocate device structure */
  dev = ddekit_simple_malloc(sizeof(struct net_device));
        [...]
  ddekit_lock_init(&dev->dev_lock);
        [...]
  err = l4re_ma_alloc(DMA_SIZE, dma_cap, L4RE_MA_CONTINUOUS | L4RE_MA_PINNED);
        [...]
  err = l4re_rm_attach((void**)&vaddr, DMA_SIZE, L4RE_RM_SEARCH_ADDR, dma_cap,
        [...]
  err = l4re_ds_phys(dma_cap, 0, &paddr, &psize);
        [...]
  err = ddekit_request_io(dev->ioport_addr,dev->ioport_size);
        [...]
  ddekit_interrupt_attach(dev->irq,0,NULL,(void*)net_rx,NULL);
        [...]
  ddekit_interrupt_enable(dev->irq);
        [...]
void NetReceivePacket(void *Priv, void *Buffer, uint32_t Size)
{
[...]
int main(void)
{
  int err;

  ddekit_init();
  ddekit_pci_init();
        [...]
```

Figure 3.6: Template code snippet

4 Evaluation

In this chapter I present the evaluation of the RTL8139 Windows driver ported to TUD:OS. The test hardware consists of an Intel Pentium D 3.0 GHz CPU with 1.5 GB of RAM. The used network card is the TRENDnet TE100-PCIWIN network adapter [TRE12] based on an RTL8139D chip.

I measured the bandwidth and CPU utilization using Iperf [GW12]. Iperf is a tool that works as a client and a server. The server is implemented in the target system's environment and the client operates from another computer. On the same ethernet network the client asks for a TCP connection and sends packets to the server.

I tested 3 different software stacks on the hardware.

1. Archlinux's Live Distribution [VG12] released August 2011
2. TUD:OS with the developed RTL8139 network driver described in Section 3.4
3. TUD:OS with Ankh and the RTL8139 Windows driver ported to Linux by the RevNIC authors as described in Section 3.1

The Iperf server runs on top of all test scenarios. The application waits for incoming TCP connection requests on port 5001 (default port), confirms it and waits for new requests. In Scenario 2 and Scenario 3 the underlying software stack consists of Fiasco.OC [TD12](kernel), Sigma0 (root-pager), Moe (root-task), and the I/O server.

In Scenario 2, the Iperf server runs on top of this stack containing DDEKit and the driver. In Scenario 3, two applications run on top of the software stack. The Ankh server and the Iperf server. On startup Ankh searches for network interface controllers and starts the corresponding driver. In this case dde-linux2.6 is used to interface between the RTL8139 Linux driver and DDE. The Iperf server connects to Ankh and communicates with the driver via shared memory. Scenario 2 and Scenario 3 use LwIP [Dun] as TCP/IP stack.

4.1 Maximum throughput

In this test case Iperf sends 100MB data. I gathered the bandwidth reported by the client and repeated the test 29 times. This number of tests gives a solid overview about variations of the maximum throughput.

The bars in Figure 4.1 indicate the average throughput and error bars show the variation of all 30 tests. The topmost bar describes Scenario 1, the middle bar Scenario 2 and the bottom bar Scenario 3. The Linux driver running in Archlinux is very constant in it's results as compared with the other implementations. All applications reached over 90 Mbits/sec and therefore saturate the 100 Mbit/s network interface controller.

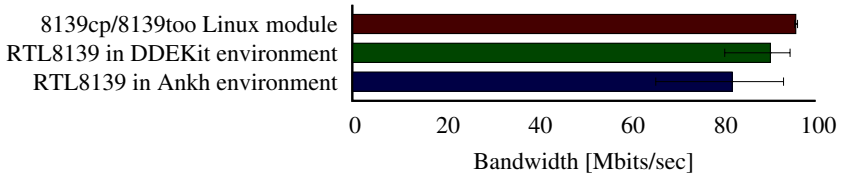

Figure 4.1: Throughput

The varying results for both TUD:OS drivers can be attributed to lwIP. While monitoring the test executions with Wireshark, there were a huge amount of TCP DUP/ACK packets on the wire due to a problem with sequence numbers which is already reported on the lwIP user mailing list [lwi10].

4.2 Segment size dependent throughput and CPU utilization

The next test case was about how much throughput can be reached and how much CPU utilization is needed depending on the maximum packet size. Therefore, Iperf's overall data was still set to 100MB and the maximum segment size (MSS) option was used to generate packets with size from 100B to 1400B incremented in steps of 100. On Linux I used *top* to monitor the CPU utilization and on TUD:OS I implemented this functionality myself. I extended the Iperf server application with a thread. This thread computes the CPU utilization by calculating the difference between 1 and the quotient of the scheduler idle time and the Iperf server's process time. The results are shown in Figure 4.2 and Figure 4.3.

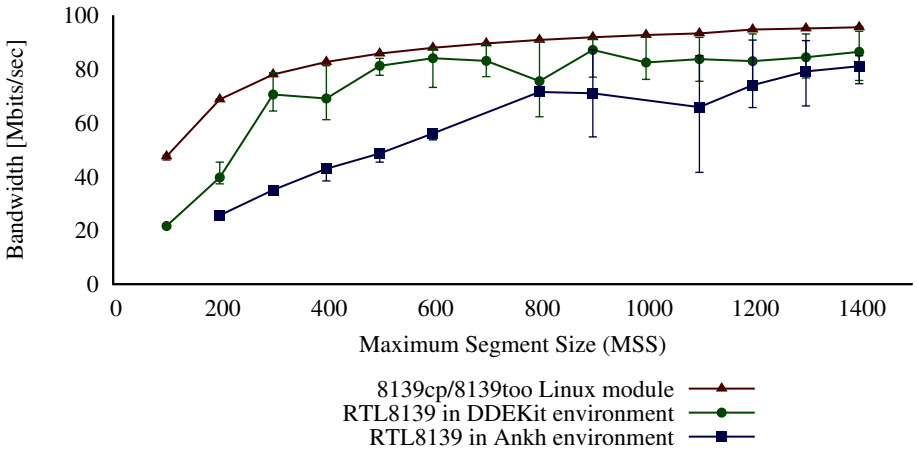

Figure 4.2: Throughput per maximum segment size

As can be seen in Figure 4.2, the TUD:OS drivers show a much larger variance in the obtained throughput. While they reach Linux results in the best cases, we can see lower average throughput. As shown in Figure 4.3, the CPU utilization of the RTL8139

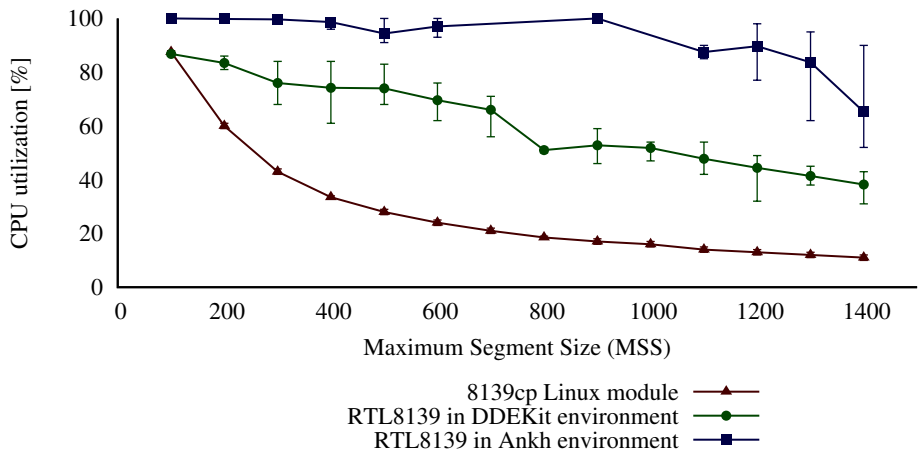

Figure 4.3: CPU utilization per maximum segment size

driver on DDEKit monotonically drops starting at about 88% and finishing at about 39%. The original Linux driver starts at quite the same CPU utilization but drops in steeper steps until the maximum segment size equals 500 Bytes. Due to the fact that Ankh's shared memory implementation polls for new data chunks, the CPU utilization is always nearly working to capacity. Results in Figure 4.2 are missing for 700 and 1000 Bytes MSS and in Figure 4.3 additionally for 800 Bytes MSS.

The main reason is the different TCP/IP stack implementation. TUD:OS drivers use lwIP in contrast to the Linux TCP/IP stack. During the test with TUD:OS, Wireshark indicated the same TCP DUP/ACK problems as in the test before. Also, TUD:OS is a microkernel operating system whereby drivers run in user mode. Therefore, many context switches need to be performed during execution which is not necessary for a monolithic kernel operating system such as Linux.

5 Conclusion and Outlook

In this thesis I used RevNIC to overcome the problem of porting device drivers to TUD:OS with the focus on maintenance effort. I based my work on DDE, a platform-independent software layer for device drivers, and RevNIC, an analysis tool and code generator for network drivers.

In Chapter 3 I described the use of RevNIC and its work flow towards a network device driver template for DDEKit. It turned out that I could not produce usable code with RevNIC. I described the errors and difficulties in Section 3.3. For further work, I used the results of the RevNIC paper [CC10] that were made available to me by the authors. I ported the generated Windows RTL8139 network device driver to DDEKit presented in Section 3.4. In Section 3.5 I presented the template, which I extracted out of the working driver. This template needs to be created once and implements the only interface required by any network device drivers generated by RevNIC to properly work on TUD:OS.

The driver's performance was evaluated in contrast with the original Linux driver and the reverse engineered Windows driver ported to Linux on TUD:OS in the Ankh environment. The measured driver's throughput is comparable to the original Linux driver in the best case, but shows higher variations due to use of a different TCP/IP stack and a split-component architecture.

5.1 Future work

Further work should aim to automatically analyze drivers with reliable results. S^2E [CKC12] could be the tool to analyse drivers in the future because RevNIC is a never-published prototype whereof S^2E was developed. S^2E is not restricted to network drivers but currently there exists no code synthesizer which transforms the analysis results into C code. Hence, a code synthesizer for S^2E needs to be developed if S^2E should be used such as RevNIC.

Moreover, the created template only supports network drivers. Therefore, additional templates for other device driver classes should be implemented for DDEKit to reduce the effort to port drivers to TUD:OS.

Bibliography

[APJ+01] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill Framework for VM diversity. *Proceedings of the 6th Australasian Computer Systems Architecture Conference, Gold Coast, Australia*, January 2001. 26

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. *USENIX Annual Technical Conferece*, 2005. 17

[CC10] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. *5th ACM SIGOPS/EuroSys*, April 2010. 7, 9, 11, 13, 17, 18, 19, 21, 33

[CDE08] Christian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted automatic generation of high-coverage tests for complex systems programs. *In 8th Symp. on Operating Systems Design and Implementation*, 2008. 15

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and application. Technical report, Ecole Polytechnique Federal Lausanne (EPFL), Februar 2012. 24, 33

[Cor12] Microsoft Corporation. Use the Microsoft® symbol server to obtain debug symbol files. `http://support.microsoft.com/kb/311503`, August 2012. 26

[Dun] Adam Dunkels. lwIP - a lightweigth TCP/IP. 29

[Fou12] Wireshark Foundation. Wireshark. `http://www.wireshark.org/`, September 2012. 22, 23

[Fri06] Thomas Friebel. Übertragung des Device-Driver-Environment-Ansatzes auf Subsysteme des BSD-Betriebssystemkerns. Master's thesis, TU Dresden, 2006. 11, 12

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. *Proceedings of the International Conference in Computer Aided Verification (CAV 2007)*, July 2007. 15

[GW12] Mark Gates and Alex Warshavsky. Iperf homepage. `http://iperf.fr/`, August 2012. 29

[Hab12] Thomas Habets. Arping homepage. `http://www.habets.pp.se/synscan/programs.php?prog=arping`, May 2012. 23

[Hel01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur. Master's thesis, TU Dresden, 2001. 11, 12

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976. 13, 15

[Lat12] Chris Lattner. LLVM homepage. `http://www.llvm.org/`, May 2012. 15

[lwi10] [lwip-users] TCP window update and TCP DUP ACK. `http://lists.gnu.org/archive/html/lwip-users/2010-11/msg00117.html`, November 2010. 30

[Poe07] Bernhard Poess. Binary device driver reuse. Master's thesis, Universität Karlsruhe, 2007. 11, 13

[Sof12] Heaventools Software. PE Explorer homepage. `http://heaventools.com/overview.htm`, August 2012. 26

[TD12] Chair of Operating Systems TU Dresden. Fiasco.OC. `http://os.inf.tu-dresden.de/fiasco/`, September 2012. 29

[TRE12] TRENDnet. TRENDnet TE100-PCIWIN product information. `http://www.trendnet.com/products/proddetail.asp?prod=210_TE100-PCIWN&cat=14`, September 2012. 29

[VG12] Judd Vinet and Aaron Griffin. Archlinux. `http://www.archlinux.org/`, September 2012. 29

[Whe12] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, September 2012. 25