Analysis, Design and Implementation of a Printing Stack for the Open-Source ReactOS Operating System

Analyse, Design und Implementierung eines Druckerstacks für das Open-Source ReactOS Betriebssystem

Colin Finck
Matriculation Number: 314570

Bachelor Thesis
at
RWTH Aachen University
Faculty of Electrical Engineering and Information Technology
Institute for Automation of Complex Power Systems
Univ. Prof. Dr.-Ing. Antonello Monti

Supervisor: Dr. rer. nat. Stefan Lankes

Online PDF Version

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published and unpublished scripts are indicated as such. The paper has not been previously presented as an examination paper in any other form.

Aachen, September 29, 2015

Kurzfassung

Das Open-Source ReactOS Betriebssystem hat das Ziel, eine Alternative zum aktuell marktführenden PC-Betriebssystem Microsoft Windows zu schaffen, indem es volle Kompatibilität mit dafür geschriebenen Anwendungen und Treibern bietet. Aus diesem Grund müssen existierende Anwendungen (z.B. Textverarbeitungsprogramme) in der Lage sein, die etablierten API-Funktionen zum Drucken auch unter ReactOS zu benutzen, ohne diese Anwendungen neu zu kompilieren oder deren Code zu verändern.

Diese Arbeit liefert eine ausführliche Recherche zum Druckerstack des Microsoft Windows Betriebssystems als auch einen Vergleich zu anderen verbreiteten Druckerstacks. Es folgt der Entwurf und eine erste Implementierung von kompatiblen Komponenten für das ReactOS Betriebssystem, um das Drucken von vorbereiteten Daten in einer Druckerkontrollsprache unter Benutzung der etablierten Betriebssystem API-Funktionen zu ermöglichen. Dies umfasst sowohl das Drucken auf echten Drucker, die physisch an einen Computer angeschlossen sind, als auch die Verwendung virtueller Drucker zur Ausgabe des Druckergebnisses in einer Datei. Die Komponenten sind flexibel entworfen, sodass eine spätere Erweiterung um Treiberunterstützung für Datentyp-Konvertierungen, Benutzeroberflächenkomponenten und zusätzliche Druckertreiber möglich ist.

Stichwörter: ReactOS, Betriebssysteme, Drucken, Reverse Engineering

Abstract

The Open-Source ReactOS Operating System is aiming to provide an alternative to Microsoft Windows - the currently dominant operating system for Personal Computers on the market - by offering full compatibility with applications and drivers written for it. As such, existing applications (such as text processing applications) need to be able to use the established API functions to print under ReactOS without recompiling these applications or modifying their code.

This thesis provides an in-depth research on the Printing Stack of the Microsoft Windows Operating System as well as a comparison to other popular Printing Stacks. The work is followed by the design and an initial implementation of compatible components for the ReactOS Operating System to enable Printing of a prepared data stream in a Printer Control Language using the established Operating System API functions. This includes printing to real Printers physically connected to a computer as well as employing virtual Printers to write the printing output into a file. The components are designed in an extensible way to allow for a future addition of driver support for datatype conversions, user interface components and additional Printer Drivers.

Keywords: ReactOS, Operating Systems, Printing, Reverse Engineering

Contents

Abbreviations

API	Application Programming Interface
AVL	Adelson-Velski Landis
BIOS	Basic Input/Output System
CUPS	Common UNIX Printing System
DLL	Dynamic Link Library
EMF	Enhanced Metafile
GDI	Graphics Device Interface
HTTP	Hyper-Text Transfer Protocol
IDL	Interface Definition Language
IPP	Internet Printing Protocol
KD	Kernel Debugging
LCG	Linear Congruential Generator
LPD	Line Printer Daemon
MIME	Multipurpose Internet Mail Extensions
PDB	Program Database
PDF	Portable Document Format
PE	Portable Executable
PPA	Printing Performance Architecture
RCE	Reverse Code Engineering
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SMB	Server Message Block
STL	Standard Template Library
USB	Universal Serial Bus
XML	Extensible Markup Language

List of Figures

1 Introduction

ReactOS is a modern desktop operating system entirely available under Open-Source licenses. The Project is unique in the way that it aims for compatibility with all existing applications and drivers developed for Microsoft Windows. This exclusive feature among free operating systems can make ReactOS an appealing alternative to the currently dominant desktop operating system. By being distributed under Open-Source licenses, ReactOS can offer customizations and trustworthiness not possible with traditional closed-source systems.

Printing has become an essential feature of graphical desktop operating systems in the 1980 years. The introduction of affordable Inkjet and Laser Printers around the same time have turned Personal Computers into Desktop Publishing machines able to produce high-quality documents [14]. Since then, computers have mostly replaced traditional typewriters and typesetter systems.

Today, Printing is a self-evident ability of Personal Computers. A desktop operating system is expected to detect and install connected Printers automatically as well as to provide intuitive options to manage and use them. One of the common tasks of server operating systems is making a Printer available to multiple users over the network. More recently, also smartphone and tablet operating systems have added support for Printing [2]. However, the ReactOS Operating System has not offered any support for Printing yet.

Operating system support for Printing also plays an important role in document exchange. Creating a document in Adobe's popular Portable Document Format (PDF) is usually realized through a virtual Printer. Such a Printer can be used as a destination from any text processing application just like a real Printer.

1.1 Thesis Work

This thesis presents an initial design and implementation of a Printing Stack for the ReactOS Operating System.

While several Open-Source Printing Systems already exist, none of them provides compatibility to the wide range of available Windows Printer Drivers. On the other hand, hardware vendors often provide the most feature-rich drivers only for the Windows platform. Therefore, the work does not build upon an existing Open-Source Printing System, but new components are developed from scratch.

In order to achieve ReactOS' goal of full compatibility to Microsoft Windows applications and drivers, the Windows Printing interfaces are analyzed in-depth. Additionally, existing code of the ReactOS Operating System is examined to de-

termine the extent of the required implementation work. In a next step, a set of fundamental components is developed, which form an initial Printing Stack. This system is able to transmit prepared RAW data in a Printer Control Language to a locally connected Printer. However, the entire architecture is designed for a further extension by other datatypes at a later stage.

Evaluation of the written code occurs with the help of specific individual unit tests covering the implemented features.

Although Microsoft provides a publicly available documentation about their Printing implementation, it does not cover specific internals. Therefore, a special emphasis has been put on commenting and documenting the resulting code to serve as a future reference.

1.2 Special Thanks

I would like to thank Univ. Prof. Dr.-Ing. Antonello Monti and Dr. rer. nat. Stefan Lankes for offering me the unique opportunity to make ReactOS part of my Bachelor thesis. Special thanks also go to all members of the ReactOS Project who are maintaining this great Open-Source Project for several years.

Colin Finck
Aachen, September 2015

2 Basics

This chapter introduces several Open-Source software projects relevant to the thesis implementation work. It is followed by an explanation of the backgrounds and technical terms regarding Printing support in operating systems. Finally, some utilities are presented, which have been widely used for the development of the Printing Stack.

2.1 The ReactOS Project

The ReactOS Project goes back to a project called *FreeWin95* in 1996 by various software developers to create an Open-Source reimplementation of Microsoft Windows 95. With no visible progress by the end of 1997, the project was restarted in 1998 as *ReactOS* and shifted its goals towards providing an operating system compatible with the Microsoft Windows NT series under the GNU General Public License [32]. The name was chosen to express the dissatisfaction with the Microsoft operating system monopoly and provide a *reaction* to it [33].

Today, ReactOS strives for compatibility with Windows Server 2003 (also known as Windows NT 5.2) at the kernel level while applications can also make use of some functions found in more recent Windows releases [10]. Several popular Windows applications such as Adobe Photoshop or Microsoft Office are running natively under ReactOS. This also applies to several drivers for hardware components such as graphics adapters, network cards, or sound cards.

By providing this level of compatibility with a very popular operating system, ReactOS aims to become a free and Open-Source alternative to it. As a lightweight operating system, ReactOS can be a solution to keep older computers usable. Finally, the Open-Source nature of ReactOS allows for customizations not possible with Microsoft Windows. It also reduces licensing costs and ensures confidentiality in sensitive environments.

2.2 The WINE Project

The Open-Source WINE Project was founded in 1993 with the goal of running Windows applications under Linux. In contrast to typical emulator software, WINE does not simulate a full x86 processor to run an operating system on it, but instead provides a loader for binaries in the Windows Portable Executable (PE) format along with a set of reimplemented Windows libraries [9]. This allows applications to deliver

a higher performance under WINE than under x86 emulators, but this performance benefit is getting lower with advancements in x86 virtualization technology.

In contrast to ReactOS, WINE does not provide any support for Windows device drivers. The project also does not target a specific Windows version, but allows users to choose a particular Windows version WINE shall mimic. Due to similar licenses, code of fundamental Windows libraries is frequently exchanged between the ReactOS and the WINE Project.

2.3 Printing Support In Operating Systems

Operating system support for Printing is as old as the Personal Computer itself, with support for then standard Dot-Matrix Parallel Port Printers being implemented into the original IBM Personal Computer 5150 Basic Input/Output System (BIOS) of 1981 [4, 29]. As neither a concept of Printer Drivers nor a common Printer Control Language existed at that time, user programs were only able to output basic unformatted text by default. Higher sophisticated printing of different fonts or graphics required software developers to implement support for each Printer Control Language into their applications. This first changed with the introduction of the Apple LaserWriter Printer in 1985, which standardized Adobe PostScript as a vendor-independent Printer Control Language [14]. Around the same time Windows 1.0 debuted, featuring a first Printing Stack consisting of a Print Spooler along with a set of Printer Drivers for converting Graphics Device Interface (GDI) output into different Printer Control Languages [13]. Instead of talking directly to the Printer, applications now just needed to call GDI functions for printing out a document. Usually, these are the same functions already used for displaying graphics and text on the screen. The Spooler is responsible for enabling non-blocking access to a single shared Printer by multiple applications.

With Windows, Mac OS X, and Linux evolving into the three popular operating systems these days, two Printing Stacks remain. These are described in the following sections.

2.3.1 Microsoft Windows Printing Stack

The Printing Stack of the Microsoft Windows Operating System is unique in the way that some higher level components maintain compatibility with all previous Windows versions. At the same time, lower level parts follow the latest principles of modern operating systems. A schema of the Printing Process under Windows 2000 and later is given in Figures 2.1 and 2.3. All components involved are explained in the following. Yellow marked nodes denote components, for which compatible replacements are implemented within this thesis.

User-Mode Windows applications interact with the operating system by calling documented Application Programming Interface (API) functions from operating system Dynamic Link Library (DLL) files. To print out a document, an application

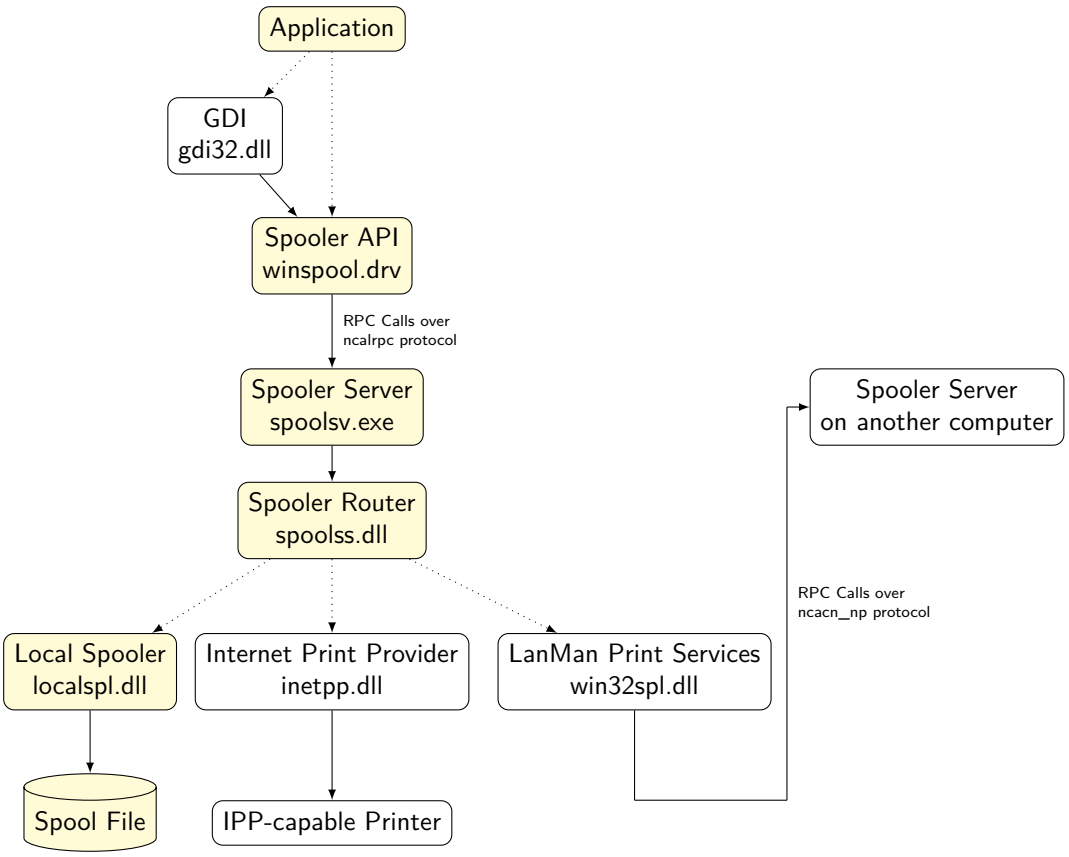

Figure 2.1: Spooling a Print Job into a file as implemented in current Microsoft Windows Operating Systems

usually begins by composing the document out of graphics and text using GDI functions (implemented in *gdi32.dll*). Afterwards, GDI serializes the drawing commands into the Enhanced Metafile (EMF) format and uses Spooler API functions to set up a new Print Job. The generated EMF data is then passed to the Print Job. If the application does not need to compose the document, but already has prepared data in a format supported by the Print Processor, it can skip the route through GDI and call the Spooler API functions directly [37].

For historical reasons, the Spooler API is implemented in a file called *winspool.drv*. Despite its different extension, this file has the same structure as other operating system DLL files. An individual instance of *winspool.drv* is loaded with every application that uses functions of the Windows Printing Stack. Its API can be categorized as follows:

- Opening handles to Ports, Print Monitors, Print Servers, and Printers (all through `OpenPrinter` [21])
- Performing further operations on these opened objects (e.g. preparing a new document with `StartDocPrinter` or retrieving information with `GetPrinter`)

- Adding, deleting, and enumerating available Forms, Ports, Print Monitors, Print Processors, Printers, Printer Configuration Data, Printer Connections, and Printer Drivers as well as their properties (`Add*`, `Delete*`, and `Enum*` group of functions)
- Receiving notifications about status changes inside the Printing Stack (`*PrinterChangeNotification` group of functions)
- Providing User Interfaces to let the user configure Printer and Print Job settings

As every application loads its own instance of *winspool.drv*, the operating system needs to provide a single service, which is loaded only once and centrally manages Printer utilization. This instance is the Spooler Server, which is implemented as a Windows Service in the module *spoolsv.exe*. Communication between *winspool.drv* and *spoolsv.exe* happens through Remote Procedure Calls (RPCs). RPC is a popular concept for enabling a process to call a function in another process, even on another computer, without writing any network-specific code. Almost every *winspool.drv* function performs an RPC call to the matching counterpart function in the Spooler Server. RPC calls are further discussed in Section 2.4.

Accepting RPC calls of all users requires the Spooler Server to be a high-privileged process. This bears some security risks as a possibly vulnerable RPC call could be used to let a low-privileged client run code in the security context of the high-privileged Spooler Server. To mitigate this possible attack, the Spooler Server employs the concept of *Impersonation*. The Impersonation feature of Windows allows a thread to temporarily drop its high privileges by switching to the security context of another user [24].

In the case of the Spooler Server, every RPC call is implemented as follows: First of all, the Spooler Server impersonates the calling client. Afterwards, a matching function in the Spooler Router is called (implemented in *spoolss.dll*). The Spooler Router offers such a counterpart to each RPC call. Finally, the security context of the Spooler Server is restored and the RPC call returns. This ensures a clean separation between code running in the high-privileged Spooler Server security context and code running in the low-privileged user context. The entire process is exemplified for a `WritePrinter` call in Figure 2.2. The full listing for the implemented ReactOS RPC server function can be found in Appendix A.2.

The Spooler Router derives the name from its principal task, namely routing an incoming function call to one or more Print Providers. These Print Providers also offer a counterpart to each function implemented in the Spooler Router. Due to the nature of the functions, there are basically three ways how a Spooler Router function is implemented:

- Subsequently route a function call to every available Print Provider until one of them indicates success. This is done for e.g. the `OpenPrinter` function to determine the Print Provider that can handle the respective Printer.

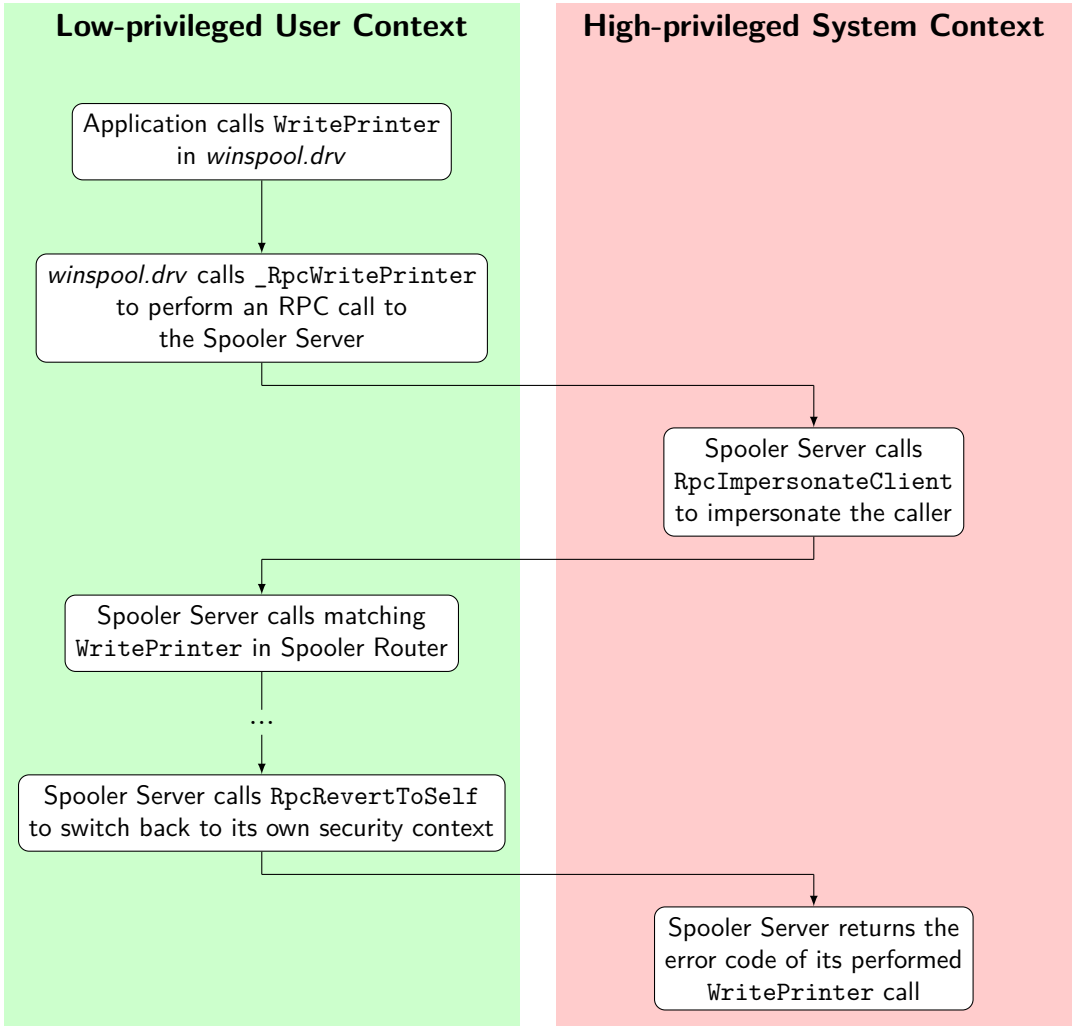

Figure 2.2: Using Impersonation to switch between security contexts while processing an `WritePrinter` call

- Directly route the function call to the Print Provider, for which a previous `OpenPrinter` call succeeded. This is done for all functions accepting a handle returned by `OpenPrinter`.
- Route the function call to all available Print Providers and collect the returned information. This is done for e.g. `EnumPrinters` to retrieve information about all available Printers.

Windows ships with these Print Providers by default:

- The Local Spooler (implemented in *localspl.dll*) handles Printers locally connected to the computer.
- The Internet Print Provider (implemented in *inetpp.dll*) forwards calls to Remote Printers using the Internet Printing Protocol (IPP).

- The LanMan Print Services (implemented in *win32spl.dll*) are used when accessing Remote Printers shared by another Windows computer.

Another Print Provider (*nwprovau.dll*) for accessing Printers on Novell NetWare servers used to be available, but became largely irrelevant due to the demise of the NetWare Operating System. It has finally been removed in Windows Vista. The extensible architecture enables third-party vendors to ship additional Print Providers for supporting other protocols.

In the following, we only take a look at the Local Spooler. The Local Spooler finally does the real work of managing the Print Queues for all locally connected Printers instead of passing on the received function call to another module. Synchronous communication between computers and Printers can be a major bottleneck. Therefore, the Local Spooler first writes the received data to print into a so called Spool File and then starts a thread to transmit that Spool File to the Printer. This enables the users to continue their work in the application while the Local Spooler transmits the Spool File to the Printer in the background. Without this two-stage process, the application would be blocked until the last page of the entire Print Job has finished printing.

The started thread now loads the Print Processor (implemented in *winprint.dll* since Windows Vista, previously part of *localspl.dll*), which is responsible for reading the print data from the Spool File and applying Print Job specific settings. These include settings like Multiple Copies, Collation, Reverse Printing, Duplex Printing or N-up Printing. Available options in this regard highly depend on the datatype of the print data. A Printer vendor can provide its own Print Processor to support additional options instead of using the Windows default one.

If the datatype is RAW, the print data is assumed to be of a Printer Control Language and the Print Processor simply passes it on without any further processing. For the EMF datatype, Print Job specific settings are applied first before the print data is converted into Printer Control Language. The conversion is performed using a Printer Graphics DLL supplied by the Printer vendor, which heavily uses functions from GDI to convert the EMF data. The Printer Graphics DLL is also called the actual Printer Driver, because it is the only component specific to the Printer in this process.

In a next step, Spooler Router functions are used to transmit the print data in a Printer Control Language to a Print Monitor. Two types of Print Monitors exist: Language Monitors and Port Monitors. A Language Monitor is written for a specific Printer Job Language to communicate bidirectionally with the Printer. This communication enables the Monitor to receive detailed Printer status information or add control codes to the print data before passing it to the Port Monitor. Control codes can be used to set a wide range of options. One example is switching between multiple Printer Control Languages supported by the Printer.

If the Printer does not require such control codes, the Language Monitor can be skipped and the print data directly flows to the Port Monitor. By default, Windows

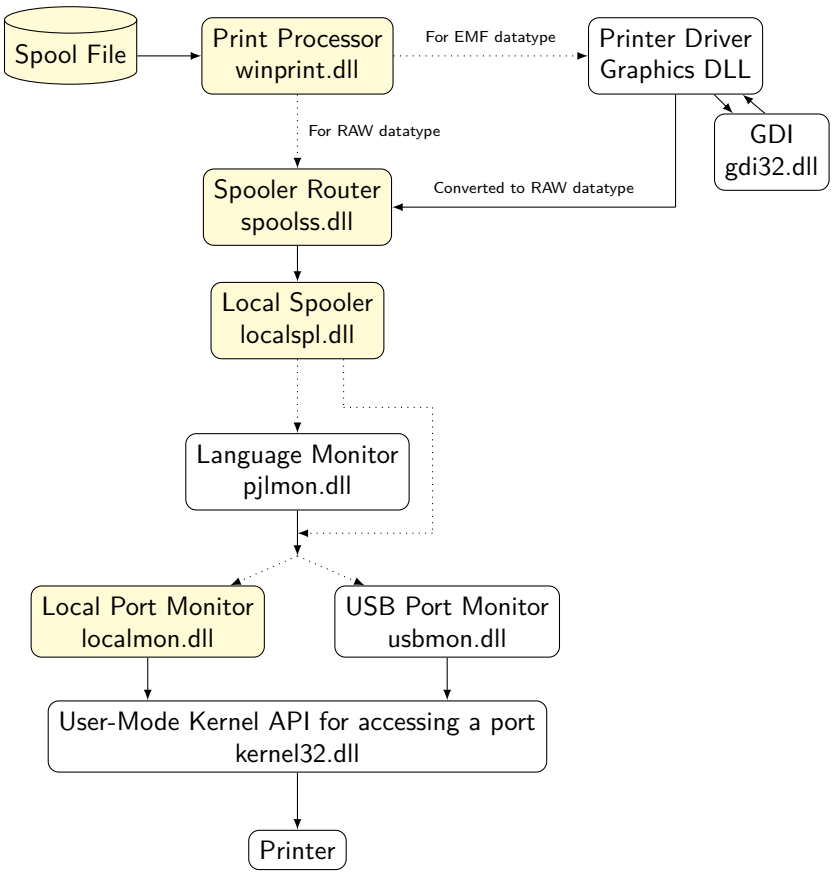

Figure 2.3: Printing from a Spool File as implemented in current Microsoft Windows Operating Systems

ships with a Language Monitor that implements the HP PJL Printer Job Language [20].

A Port Monitor is responsible for managing unidirectional data output to a physical computer port. Depending on the type of port, this can range from simply opening the port through a kernel function (like Parallel Ports) to performing a complex wireless detection sequence (for Infrared Printers). The following Port Monitors are shipped with Windows:

- The Local Port Monitor (implemented in *localmon.dll* until Windows NT 4.0, part of *localspl.dll* since Windows 2000) manages Parallel, Serial, and Infrared Ports as well as redirecting the printing output into a file.
- The USB Port Monitor (implemented in *usbmon.sys*) manages Printers connected to a Universal Serial Bus (USB) port.

This architecture again provides support for additional Print Monitors. Such extensibility is heavily used by third-party companies. For example, Adobe has

implemented a PDF Port Monitor for a virtual PDF Printer that converts the print data into a PDF file [1].

2.3.2 Common UNIX Printing System (CUPS)

The Common UNIX Printing System (CUPS) was developed and released by Michael Sweet in 1999 to address the lack of a standard printing interface in UNIX-based operating systems (such as Linux) at that time [36]. Previously, there were two competing systems, namely the System V Printing System (lp) and the Berkeley Printing System (lpr), which were incompatible to each other and only supported text and PostScript printing. Printing other formats required third-party tools to account for the vast majority of available Printers.

CUPS has quickly emerged as the de-facto standard Printing Stack under Linux, with User Interfaces being available for the two major desktops KDE and GNOME [27] [5]. CUPS has also been adopted by Apple in 2002 to serve as the Printing System for Mac OS X 10.2 and later versions [7].

The architecture of CUPS is depicted in Figure 2.4.

An application can initiate a Print Job with CUPS in three different ways. The most common one is using the `cupsPrintFile` function (or a similar one) of the CUPS C API to print a file of a known filetype. The API function then initiates a Request over the IPP protocol to a CUPS instance on a local or remote computer. As IPP is a documented plaintext protocol based on Hyper-Text Transfer Protocol (HTTP), an application can also easily generate a similar IPP request itself and transmit it to the CUPS Daemon without using the CUPS API. Finally, CUPS also provides the *cups-lpd* daemon to provide compatibility with older applications using the Line Printer Daemon (LPD) protocol of the Berkeley Printing System. cups-lpd translates incoming LPD Requests to the IPP protocol using the CUPS API.

In all three cases, the Print Request arrives at the CUPS Daemon. Depending on the Print Job settings and Printer utilization, the Job is processed immediately or scheduled for later. When processing the Job, CUPS first checks its filetype. PostScript files can instantly be forwarded to the *pstops* program while other filetypes need a conversion to PostScript format by the so called *prefilter* first. CUPS can support any filetype, for which a converter program to PostScript exists, like plaintext files, image formats or web pages. Filetypes and converter programs are determined using the system-wide Multipurpose Internet Mail Extensions (MIME) database available on every UNIX-based operating system. In a next step, the converter program is called and its output is fed to the *pstops* program.

The *pstops* program expects a PostScript input file and applies Print Job specific settings like N-up Printing or extracting a page range to print. It also normalizes the input to account for the paper format of the target Printer.

The type of Printer used decides about the next step. If the target Printer understands PostScript, the output from *pstops* can directly be submitted to a CUPS Backend. Otherwise, the next step depends on whether CUPS natively provides support for the target Printer. If it does, the output is first converted to a CUPS

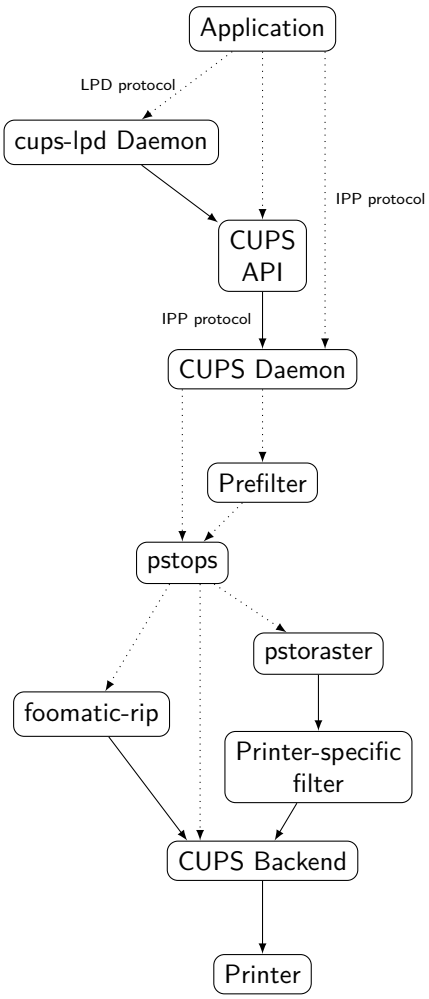

Figure 2.4: Architecture of the Common UNIX Printing System

raster image file format using the *pstoraster* tool and then sent to a Printer-specific filter, which generates data in a Printer Control Language from it. Finally, that data is submitted to a CUPS Backend as well. If CUPS does not natively support the target Printer, a solution may still be found in the third-party Foomatic Project. The Foomatic Project provides a set of programs that convert PostScript data to various Printer Control Languages without utilizing an intermediate raster format. If the Printer is supported by the Foomatic Project, *pstops* passes its output to the *foomatic-rip* program, which in turn outputs data in a Printer Control Language and submits it to a CUPS Backend [26].

CUPS provides several backends for transferring the received data to the actual Printer. The type of backend used depends on the port and location of the target Printer. For example, this can be a Printer connected to a Parallel, Serial, or USB port as well as a Remote Network Printer communicating over e.g. IPP, Server Message Block (SMB) or JetDirect protocol.

2.3.3 Comparison Of Both Systems

A neutral comparison of both Printing Stacks is not possible, because they were designed in different epochs, on different operating system architectures, and with different compatibility targets and different third-party support strategies in mind. However, some major differences between both systems are outlined in this section.

The Windows Printing Stack allows third-party Printer vendors to add support for their Printer by implementing a custom Printer Graphics DLL. This DLL file is then responsible for converting data from EMF format to a Printer Control Language. On the other hand, CUPS provides several ways to add support for a new Printer. The program for converting data to Printer Control Language could be a filter program that takes the CUPS raster data as input. It could also be a Foomatic plugin that directly operates on PostScript input data. This offers developers more flexibility, but shifts the complexity to the user, who has to know what type of CUPS Printer support the Printer vendor implemented. Based on this information, different steps need to be taken to install the Printer.

CUPS follows the UNIX principle of one program per task, so it divides its processing into several individual programs. From one processing step to another, process parameters are constructed, a new process is created and its parameters are parsed. Afterwards, the print data output of the previous process is copied to the new process by writing it to its standard input handle. The whole procedure requires process creation on the underlying operating system to be a relatively cheap operation in order to make the performance loss involved negligible. Some overhead is also added due to the construction and parsing of process parameters. On the other hand, the Windows Printing Stack interfaces between different components by using C function calls and passing memory pointers. This works with no additional overhead and does not add implications to the performance of the underlying operating system.

One may also expect severe scalability differences when comparing both Printing Stacks at serving Print Requests for a high number of clients. Traditionally, the Print Server needs to perform the expensive task of converting the incoming print data to Printer Control Language for all clients. When a CUPS instance is configured on both the client and server though, the CUPS architecture provides a way to offload data conversion to the clients and only send converted data in a Printer Control Language to the target Print Server. Starting with Windows Vista, Microsoft has also introduced this feature in the Windows Printing Stack under the name *Client-Side Rendering* [18]. Therefore, both Printing Stacks perform equally well at serving a high number of clients nowadays.

2.4 Remote Procedure Call

A Remote Procedure Call (RPC) is an interprocess communication concept to let programs execute functions in other programs, which are typically running on a different computer. The required information about the function call and its pa-

rameters are transmitted over the network. RPC is widely adopted due to the fact that it abstracts these network communication details away from the developer. This way, a remote function call be done just as easy as a local one and the developer does not need to write network-specific code.

The first popular implementation was the Open Network Computing Remote Procedure Call for UNIX-based operating systems proposed by Sun Microsystems in 1984 [6] (also called *Sun RPC*). Nowadays, popular implementations include Java's object-oriented Remote Method Invocation (RMI) as well as Microsoft's implementation called *MSRPC* integrated into Windows. Latter one is further illustrated in this section.

MSRPC is derived from the Open Software Foundation's Distributed Computing Environment DCE/RPC implementation and first appeared in Windows NT 3.1 [3]. Among other details, it extends the DCE implementation with additional parameter types and transport protocols while still maintaining backwards compatibility with it [16].

To make use of MSRPC, a developer has to define an interface through a supplemental file written in Interface Definition Language (IDL). This file contains prototypes for all functions to be called remotely, annotated with details about their input and output parameters as well as the length of the data transferred through each parameter. Afterwards, Microsoft's MIDL compiler is used to generate C code out of an IDL file, separately for the client and server applications. The developer can then write both applications communicating with each other. Finally, each application is linked against Windows' *rpcrt4.dll* library, which is responsible for the underlying network communication happening in the background.

Every function defined in the IDL file can be called in the client application and the call arrives at a function with the same name defined in the server application. This happens in the following way:

1. The client calls a function defined in the interface. This actually calls the generated client code by the MIDL compiler.

2. The generated code composes a network message out of the function name and all given parameters. This process is called *Marshalling*.

3. The network message is sent by the client and received by the server application.

4. The generated code of the server application reconstructs the function name and its parameters out of the message. This process is called *Unmarshalling*.

5. With all this information, the generated code calls the respective implemented function in the server application.

MSRPC builds the basis for several Windows core components, one of them being the Print Spooler. As illustrated in Figure 2.1, RPC communication happens locally

between the Spooler API in *winspool.drv* and the Spooler Server in *spoolsv.exe*. It also happens remotely from the LanMan Print Services component in *win32spl.dll* to a Spooler Server on a different computer. Two different RPC protocols are used here, which are explained in the following:

- **ncalrpc**
  The Network Computing Architecture Local Remote Procedure Call Protocol represents the most efficient way to perform an RPC call, when client and server application both reside on the same computer [22]. This is the case for *winspool.drv* and *spoolsv.exe*.

- **ncacn_np**
  The Network Computing Architecture Connection-Oriented Named Pipe Protocol supports an RPC call between different computers relying on Named Pipes for the network transport [16]. Named Pipes represent a high-performance communication method integrated into Windows. This protocol is well suited for communication between *win32spl.dll* and a remote *spoolsv.exe* instance.

A deeper look into RPC would go beyond the scope of this thesis. Refer to [12] for further information on this topic.

2.5 Reverse Engineering Tools

Reverse Engineering in Computer Science or Reverse Code Engineering (RCE) is the process of examining an existing software or technology in order to gather enough details for an own reimplementation that is compatible to the original. Various legal methods backed by court decisions exist to perform this process without infringing copyrights. These are commonly referred to as *Clean-room Reverse Engineering*.

Popular realizations of Reverse Engineering include the creation of a 100% compatible reimplementation of the IBM PC BIOS by Compaq Computer Corporation for its own computers in 1982. This step was fundamental to the computer industry as it paved the way for a large upcoming market of IBM-compatible computers. Reverse Engineering techniques have also been used during the development of the LibreOffice Productivity Suite to enable compatibility with Microsoft Office file formats. The LibreOffice implementation is available for more operating systems, increases competition on this market and avoids a vendor lock-in to products of a single company.

With Microsoft Windows being the most popular closed-source operating system, several advanced tools for Clean-room Reverse Engineering under Windows exist. Some of them have been used during the thesis work and are presented in the following.

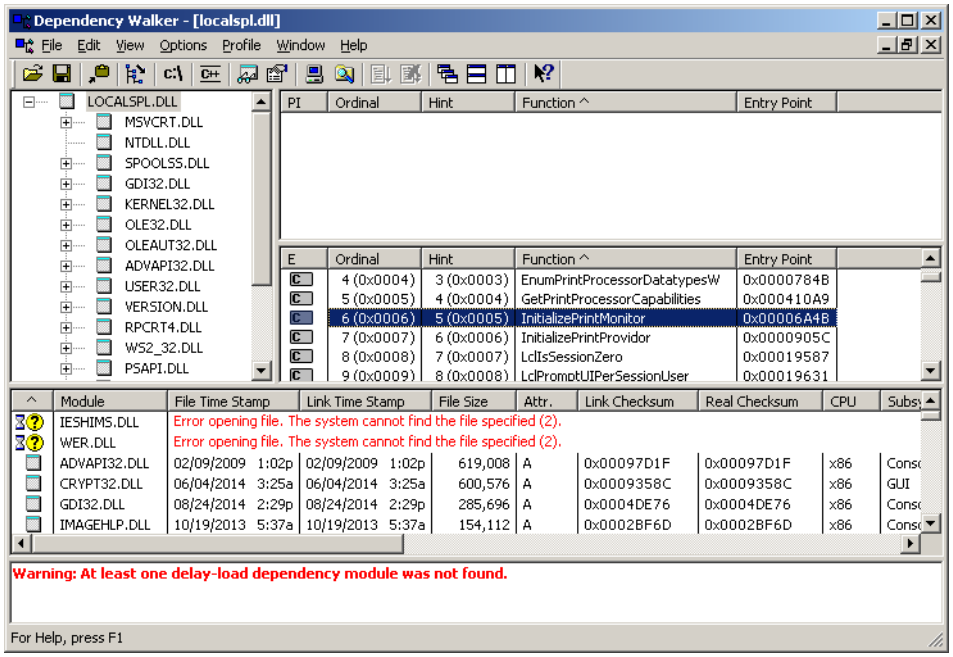

Figure 2.5: The Dependency Walker tool used to analyze the *localspl.dll* of Windows Server 2003

2.5.1 Dependency Walker

The Dependency Walker tool provides an overview of the Dynamic Link Libraries (DLLs) referenced by a module as well as the exported and imported API functions from this module. It thereby offers a good entry point into figuring out the purpose of a module and its dependencies with other modules. The tool is bundled with a number of Microsoft development products and also available as a free download from www.dependencywalker.com. A screenshot of the Dependency Walker user interface is given in Figure 2.5. It clearly shows the exported API functions `GetPrintProcessorCapabilities`, `InitializePrintMonitor`, and `InitializePrintProvidor` of Windows Server 2003's *localspl.dll*. From this fact, one can conclude that this *localspl.dll* integrates the three distinct components Print Monitor, Print Processor, and Print Provider into a single module.

2.5.2 GNU strings

The GNU strings utility reads any binary file and outputs all found sequences of printable characters. It supports interpretation of the characters as ANSI and Unicode strings, which is a requirement for analyzing Windows applications that often use both character sets. The tool reveals many interesting strings contained in Windows modules, for example:

- Registry keys used to store the module's settings

- Paths to other related files
- Template data used by the module to fulfill its tasks
- Error messages (useful to get an idea what tasks are fulfilled by the module)

GNU strings is part of the Open-Source GNU Binutils package and shipped with many Linux distributions [8]. A Windows version of the tool is part of the ReactOS Build Environment.

2.5.3 Rohitab Batra's API Monitor

The API Monitor by Rohitab Batra is a freely downloadable tool from rohitab.com. It offers an efficient graphical user interface for monitoring the API function calls of selected applications. For API functions known to the tool, the supplied parameter values are extracted and passed structures are decomposed. The application can be enhanced to monitor additional API functions by writing simple function definition files in Extensible Markup Language (XML). A screenshot of the tool is given in Figure 2.6. This one reveals that Windows' Spooler Server actually calls the Spooler Router function `AddJobW` when it receives a `StartDocPrinter` call over RPC, despite the existence of a matching `StartDocPrinterW` function in the Spooler Router. Passed parameters before and after the call are extracted and subsequent API calls recorded.

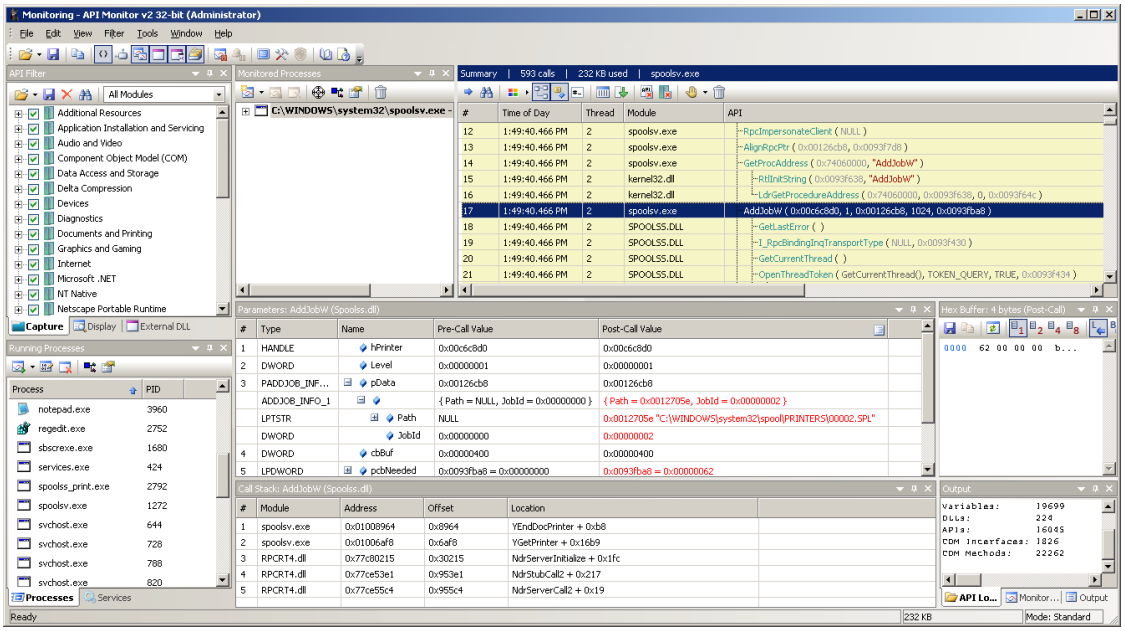

Figure 2.6: Rohitab Batra's API Monitor used to analyze an RPC call to the Spooler Server in Windows Server 2003

2.5.4 WinDbg

WinDbg is a debugger offered as a free download by Microsoft. It supports debugging User-Mode and Kernel-Mode applications and is generally the debugger of choice for Windows driver developers due to its tight integration into the Microsoft development environment. Kernel-Mode debugging can happen on the local system or on a remote computer that is connected through a Serial, FireWire, USB or Ethernet cable. In a next step, Windows is booted with the Kernel Debugging (KD) Protocol enabled to let the debugger connect and afterwards break into every component of the operating system at any time. This Protocol has also been adopted by the ReactOS Operating System to provide a debugging experience comparable to Windows. That means, the same WinDbg application can be used to debug ReactOS just like a usual Windows environment.

WinDbg can retrieve information from Program Database (PDB) files to provide single stepping through C/C++ source code files and detailed symbol information. PDB files are either generated by the Visual C++ compiler (for self-written applications) or downloaded from the Microsoft Symbol Server (for closed-source Windows components). While latter ones do not reveal any source code of Windows components, the closed-source PDB files make WinDbg aware of some variables and function names. On top of this, WinDbg comes with a handful of extensions that provide several abstract views on the operating system state. For example, this encompasses loaded processes and threads, bluescreen analysis, and hardware status information. A typical WinDbg screen is shown in Figure 2.7. With all these possibilities combined, WinDbg provides quite a powerful tool to gather information about any software under Windows or single step through it.

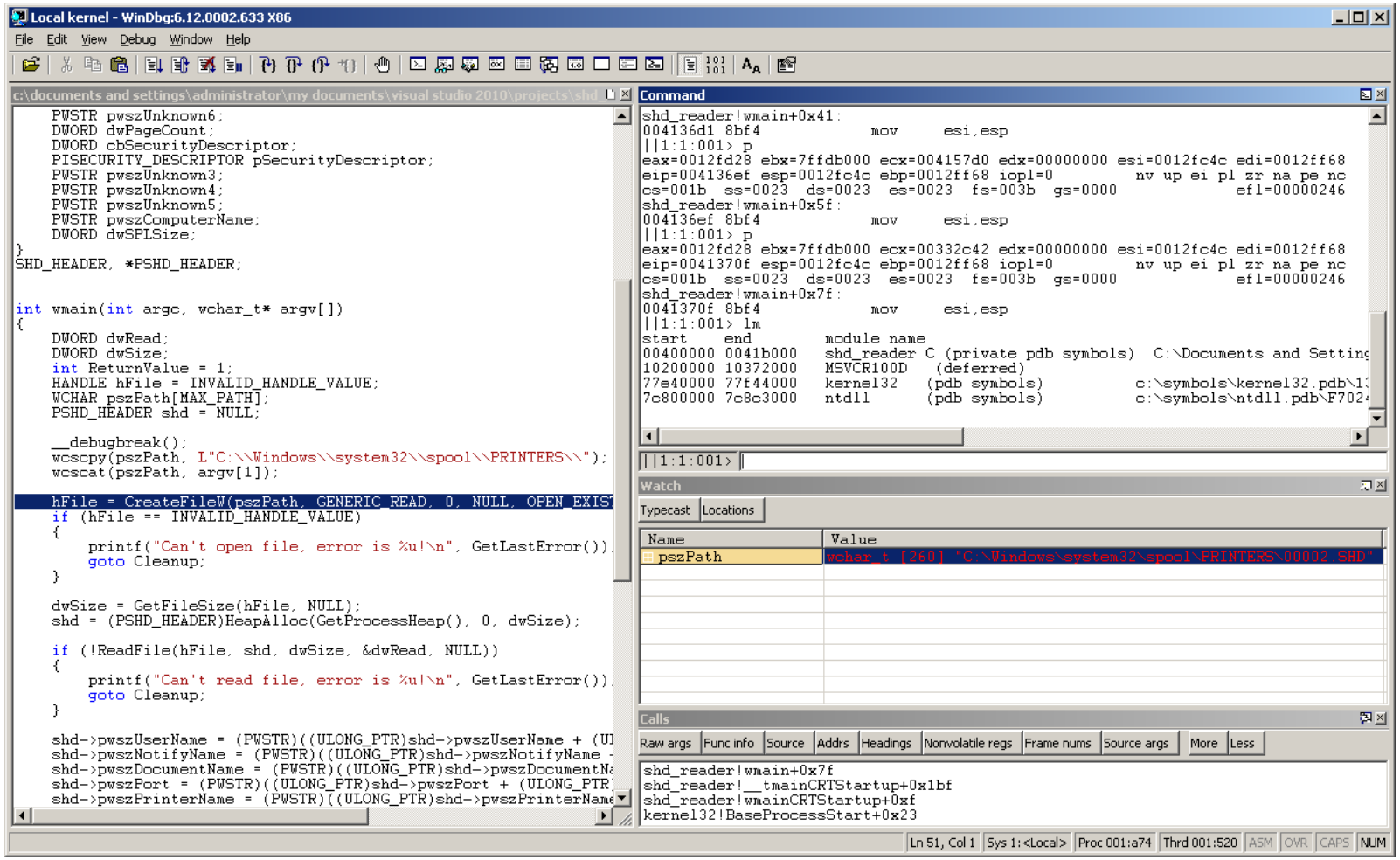

Figure 2.7: WinDbg debugging a User-Mode application with loaded PDB information in Windows Server 2003

3 Implementation

This chapter details the conducted research of the required components for the Printing Stack as well as their actual implementation. The developed components build up the foundations for a ReactOS Printing Stack that maintains compatibility with Windows API functions.

3.1 Examination Of Available Code

As a first step, the ReactOS codebase was examined to to figure out available and unavailable components as well as the quality of their code. Prior to this work, the project's source tree merely provided a basic skeleton for a Spooler Server along with some Printing components imported from the WINE Project. An inquiry to the original developer revealed that the Spooler Server skeleton was only added to let applications check the existence of that particular service. The service did not serve any meaningful purpose, so it could be safely removed and implemented from scratch during the thesis work.

A further inspection of WINE's Printing components led to the conclusion that their code could not be reused either. WINE only provides a compatible replacement of *winspool.drv* to account for many standard cases of Printing applications. But instead of forwarding calls to the remaining components of the Windows Printing Stack, WINE's *winspool.drv* translates incoming Print Requests to CUPS commands. WINE's Spooler Server is only a basic service skeleton as well and additional DLL files like *spoolss.dll* and *localspl.dll* are unimplemented in essential parts. Due to these reasons, the WINE Printing components were removed from the tree as well.

On the plus side, the ReactOS ecosystem has matured enough to provide a working RPC Server, compatible kernel functions and a Parallel Port Driver. This allowed a clear focus on printing-related components during the thesis work.

The only other notable Open-Source project sharing similar goals is the Samba Project, which implements several components to let UNIX computers interact in a network with Windows machines. These components also include support for Network Printing. But due to Samba's focus on UNIX systems, only some interface information could be used. No related code for the implementation of a ReactOS Printing Stack has been offered by the project.

After the ReactOS codebase had been completely examined, development on the new Printing Stack components could start.

3.2 Defining The Interfaces

Implementation on a ReactOS component providing compatibility with a Windows one begins by defining its API functions in a so called *SPEC* file. The SPEC file format was invented to account for the differences in linker information file formats between the GNU Linker and the Microsoft Linker. Additionally, one can denote a function as a *stub* in a SPEC file and that function will be exported with a simple entry point. If an application calls such a stub function, the call is reported through a text message, but nothing else happens. This still allows other features of the application to work properly. Without stub functions, the application may crash or not even load at all. SPEC files are processed by the *spec2def* utility during the ReactOS build process, which converts the information into a linker information file suitable for the used linker and generates the entry points for the stub functions.

The names of all provided API functions in *winspool.drv* and *spoolss.dll* could easily be determined by opening the matching DLL files from Windows Server 2003 in Dependency Walker and inspecting the Exports section. Consequently, parameters for several basic Printing APIs were looked up on the web. As these functions are fundamental to any application making use of Printing, the MSDN Website offers a detailed documentation on them. After linking the documentation from both sources together, the SPEC files for *winspool.drv* and *spoolss.dll* were written. Fundamental functions were added in all details while most of the other APIs were defined as stubs to allow a later implementation when necessary.

The Spooler Server process required a radically different approach to gain compatibility. Its interfaces are not based on regular function calls, but use RPC calls defined in IDL files. These cannot simply be revealed by a tool like Dependency Walker. Instead, they usually require monitoring of the RPC functions and reconstruction of the names and parameters.

The Samba Project has done this work for its Network Printing support and provides a corresponding IDL file. A similar research has also been independently conducted in [15]. Furthermore, Microsoft has begun to document the Print System RPC Interface in 2007 and now provides a freely usable IDL file covering a subset of the RPC functions [17]. By combining the information from these three sources, a detailed IDL file could be constructed for defining the ReactOS Spooler Server interfaces.

3.3 Choosing A Programming Language

ReactOS components are either written in C or C++, with some performance-critical or processor-specific code written in Assembly language. Due to the portability issues involved, Assembly language is only used as a last resort though. It definitely offers no advantages for the development of a Printing Stack.

The remaining decision between C and C++ has been made based on compatibility requirements. Advantages of C++ for a Printing Stack would lie in the

object-oriented approach towards lists and strings. The Standard Template Library (STL) shipped with every C++ development environment provides container classes like `list` and `vector` as well as `string` for handling character sequences of arbitrary length. However, Windows exposes a pure C interface for its fundamental Printing Stack functions due to historical reasons. Using C++ objects internally would require conversions between both data formats in every step. This would basically lead to additional overhead and cancel out most advantages of C++ for the development of a Printing Stack.

Therefore, the choice fell on the C language for all developed components of the Printing Stack. The unavailability of standard container classes in C is outweighed by the number of library functions ReactOS provides for development in C.

3.4 Developing The Required Components

As a first step towards an adequate ReactOS Printing Stack, a small test application called *winspool_print* was written in C. It uses fundamental API functions of the *winspool.drv* component to send a file with arbitrary RAW data to a local Printer. Afterwards, the goals of the development work were defined around this application: The created ReactOS Printing Stack must implement all required APIs and components for getting *winspool_print* to work. Code shall be written in a compatible and flexible way that easily allows a further addition of the not yet implemented features.

In particular, this led to the following components being developed:

- **winspool.drv**
  The Spooler API in *winspool.drv* implements all character encoding dependent functions for the default single-byte character set and double-byte Unicode. For the beginning, only some Unicode functions have been implemented, because these are recommended for new applications. Future implemented single-byte character set functions will simply convert between both encodings and then call a corresponding Unicode function. Besides, more advanced Spooler API functions not required for the *winspool_print* tool were added as simple stubs for now.

- **spoolsv.exe**
  The Spooler Server directly builds upon the IDL file, which defines its complete RPC communication interface. As a result, all 96 RPC calls need to be implemented as functions inside *spoolsv.exe*. Since many of them are unrelated to getting the basic *winspool_print* tool to work, some were just implemented to return the error code `ERROR_INVALID_FUNCTION` when being called. The others correctly perform Impersonation and pass the call to a Spooler Router function.

- **spoolss.dll**
  The Spooler Router has been implemented with the same subset of API functions that were considered for *winspool.drv*. Apart from this, the DLL also provides fundamental Impersonation and Memory Management functions to all Printing components. Impersonation has been discussed in Section 2.3.1.

  Due to the wide usage of these APIs, they received intensive *Black-Box Testing*. That means, the functions were called with defined input data under Windows and their output was recorded. Based on the correlations between input and output, hypotheses were made and verified with further similar tests. Finally, enough information has been gathered to write compatible substitution code.

- **localspl.dll**
  For sending RAW data to a Printer, the Local Spooler in *localspl.dll* needs to manage Printers, Print Jobs, Print Monitors and Print Processors as well as a list of Ports managed by Port Monitors. Support for Forms, Printer Drivers as well as their configuration data can be added at a later stage.

  In contrast to the Windows counterpart of *localspl.dll*, the Port Monitor and Print Processor parts were sourced out into individual DLL files for the ReactOS implementation. This decision logically separates such distinct components and follows the Windows Printing Architecture more closely.

- **winprint.dll**
  The variant of *winprint.dll* implemented for ReactOS contains the default Print Processor called *WinPrint*. Under Windows Server 2003, this one is part of *localspl.dll* and supports EMF, RAW and TEXT datatypes. To achieve the goal of getting *winspool_print* to work, code for processing the RAW datatype is needed. This code simply passes the data to the Port Monitor without altering it in any way. On the other hand, it has been developed flexible enough to account for a later addition of more datatypes.

- **localmon.dll**
  The Local Port Monitor is also part of *localspl.dll* under Windows Server 2003, but has been sourced out into the *localmon.dll* file for ReactOS. While the original Windows Local Port Monitor provides support for Parallel and Serial Ports along with Infrared Printers, support for the latter has not been implemented into the ReactOS counterpart yet. This decision originates from the unavailability of an Infrared Printer at the time of development as well as the added complexity that comes with a wireless link.

The implemented components are marked in yellow in Figures 2.1 and 2.3. Making them usable for the intended purpose also required an integration into the ReactOS Build System. This step is illustrated in the next section.

3.5 Integrating The Components Into The ReactOS Build System

The ReactOS Project features a Build System based on the CMake Build Automation Tool. This system supports building ReactOS under Windows, Linux, or Mac OS X using either GCC or the Microsoft Visual C++ compiler. In order to properly add the developed components to the operating system build process, several so-called *CMakeLists* files have been written. These are usual plain-text files editable with any text editor. Each of them defines a module, its type (Application or DLL), its target location in the operating system, the associated source code files, and the libraries it depends on. This information is sufficient for CMake to call the corresponding compilers and linkers to build suitable binary files.

3.6 Verification During Development

One of the basic rules when writing operating system code is to make no assumptions at any stage [11]. The developed APIs will later be called by thousands of third-party applications. This requires a developer to consider every possible case of calling a particular function and respond accordingly.

In the case of the Windows Printing Stack, a pure C API is exposed. It throws no exceptions and responds to failure cases by setting an appropriate return value and a Win32 Error Code [23]. Such error codes can be very specific and some applications may implement code paths that check for a particular error code. Therefore, developing a compatible reimplementation also requires figuring out the returned error code for each possible failure case.

The ReactOS Project is tackling this problem through *Regression Tests*. After verifying the behavior of an API function under Windows, a developer usually writes a Regression Test that calls the function with defined input data and checks that it returns the verified output. The test is then run under ReactOS to verify that the implementation of the function is compatible to the original. In a next step, regularly running this test catches cases where changes in one module accidentally break functionality in another module. Currently, the ReactOS Project runs all available Regression Tests after every commit to the source code repository. The results are then parsed, inserted into a database and later searchable online through the ReactOS Test Manager.

Writing Regression Tests for *winspool.drv* was straightforward due to the fact that this is a documented API used in every Printing application. For now, the test is only able to call deterministic informational functions though (like `GetPrintProcessorDirectory`). Many other functions require the presence of a Printer, which is not guaranteed, and they also behave differently based on the type of Printer connected. Consequently, these functions are not yet covered by the Regression Tests.

However, other components of the Printing Stack are harder to test. A prominent example is *localspl.dll*. Under Windows, the Spooler Server, Spooler Router, and

this DLL are intertwined with each other in a way that prevents loading *localspl.dll* in any other application.

Nevertheless, it was possible to verify the behavior of its Print Provider functions under Windows by using a different approach: The code for testing the function was put into a self-written DLL file. A standalone application was then launched with highest privileges in order to let it use the Windows `CreateRemoteThread` API function, which creates a new thread in the running Spooler Server process [19]. The created thread loaded the testing code from the DLL file and by running that code in a thread of the Spooler Server, it can access all functions of *localspl.dll*. Eventually, this testing method offered unique insights into the internal behavior of the Local Spooler. These were used to develop a compatible replacement and verify it against the original.

However, gathering such information from existing components is only one part of developing a Printing Stack for ReactOS. At a later development stage, the design of custom data structures also became necessary. This step is further discussed in the next section.

3.7 Designing The Data Structures

One of the Local Spooler's primary tasks is managing a changing number of Printers, Print Monitors, and Print Processors in a list. Along with every Printer comes a prioritized Job Queue. The exposed Spooler API allows to add, remove, and enumerate items from these lists. Moreover, it also allows Print Jobs to be reprioritized or to get their Job index in the list. Calling these APIs often causes lookup operations to be performed in the background.

Windows offers a variety of abstract data types ready to use in Kernel-Mode and User-Mode applications. Two of them were further analyzed with a focus on usability for the Printing components:

- **Doubly-linked Lists**
  A Doubly-linked List implemented through Windows' `LIST_ENTRY` structure is the simplest abstract data type for these tasks. It can do standard operations like additions, enumerations, and removals always in $\mathcal{O}(1)$, provided that no lookup is necessary beforehand. Lookups are an expensive operation for Linked Lists though, because they are only doable in linear time. Consequently repriorizing a Job, which involves deleting and reinserting an element at another position, needs linear time as well and shares the same $\mathcal{O}(n)$ complexity. Therefore, the focus shifted to other abstract data types.

- **Generic Tables**
  The Windows Run-Time Library also offers a data structure called *General Table*. Depending on the compile options, this one is either implemented as an Adelson-Velski Landis (AVL) tree or a Splay Tree [25]. Both are comparable in complexity, so they will not be discussed separately. The tree structures also

offer $\mathcal{O}(1)$ complexity for additions and removals on average. Lookups and consequently repriorizations are usually faster, with a complexity of $\mathcal{O}(\log n)$ on average. On the other hand, enumerations bear a higher complexity, since finding the next element of an in-order traversal is a non-trivial operation.

For usage inside the Printing components, there is another major downside compared to the `LIST_ENTRY`-based implementation of Doubly-linked Lists: Every insertion allocates a new block of memory instead of reusing the existing one. Among added overhead, this makes pointer references more complicated, because the element address is only known after the insertion. Another disadvantage is the opaque structure of Generic Tables, which allows no further extensions. Due to the already high complexity of enumerations, this makes figuring out the element index inside Generic Tables an even more expensive operation, with a complexity greater than $\mathcal{O}(n)$.

As both integrated abstract data types were not satisfying the requirements, a data structure adjusted to the needs of the Printing components has been developed.

3.7.1 Skip Lists

Skip Lists were introduced by William Pugh in 1990 as an easy to implement alternative to balanced trees [31]. They build upon the structure of singly-linked lists, but every node features an array of pointers instead of a single pointer to a next node. The array size is predefined at compile time.

Every time a node is inserted into a Skip List, a geometrically distributed random function for $p = 0.5$ is called to return a *level* for it. The level determines how many pointers to a next node are used. Consequently, it can be between 1 and the array size. In the long run, every next higher level appears only half as often as a previous one due to the geometric distribution. This results in a structure as exemplarily depicted in Figure 3.1.

Just like for singly-linked lists, the first pointer always links to the directly adjacent element. The second pointer connects all nodes that received a level of 2 or higher. This goes on for the third pointer, the fourth pointer, etc.

Such a probabilistic approach generates a data structure with lookup properties comparable to binary trees. It only involves a cheap random function to achieve this instead of enforcing complex and expensive rebalancing operations.

Lookups are implemented by starting from the head node and checking the next node on the highest used level. The code then passes all nodes on this level coming before the node it is looking for. When a final node on one level is reached, the code goes down a level and continues there. These steps are repeated until the first level and the desired node is reached. The advantageous distribution of levels across the Skip List leads to an average complexity of $\mathcal{O}(\log n)$ for lookups.

Insertions and removals in Skip Lists always need to perform a lookup first. However, the remaining operations are doable in $\mathcal{O}(1)$, so the final complexity for both operations is $\mathcal{O}(\log n)$ too.

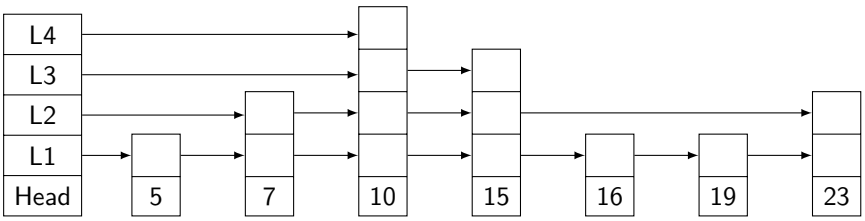

Figure 3.1: Example of a Skip List with four pointers for each node

For ReactOS Printing components, the Skip List has been implemented in a way that it does not copy the element data. Instead, it only saves a pointer to the data inside a node. This prevents overhead and keeps previous pointer references to the element intact.

The simple structure of Skip Lists also allows for three extensions not found in known implementations:

- **Reduction of comparisons**
  A lookup operation in any data structure requires multiple comparisons to be performed. To allow for every possible sorting order, the comparison is usually implemented by calling a previously specified Compare routine. As calling a function can be an expensive operation, an extension was proposed in 1990 to reduce the number of required comparisons. This is done by remembering the last compared element and never comparing the same element twice during the lookup [30]. The extension has been implemented into the Skip List for the Printing components.

- **Fast lookup of element indexes**
  A concept for looking up an element by index in a Skip List has also been proposed in [30]. It works by introducing an array, which keeps information about the distance to each other node. An exemplary Skip List with distance information is depicted in Figure 3.2.

  Maintaining these distance arrays requires slightly more algorithm complexity in the insertion and deletion functions. On the plus side, lookups can now return the element index as well while still keeping the same $\mathcal{O}(\log n)$ complexity on average. Such a lookup algorithm is given in Algorithm 1.

- **Insertion of elements at the end of the list**
  A common task in Printing is adding a new Print Job with default priority. Such jobs are always inserted at the end of the Job List. Nevertheless, the standard insertion function would perform multiple expensive calls to the Compare routine to reach the Skip List tail.

  Therefore, an optimized function called `InsertTailElementSkiplist` has been introduced for the ReactOS Skip List implementation. This function simply takes the shortest route to reach the end of the Skip List without the overhead of calling the Compare routine.

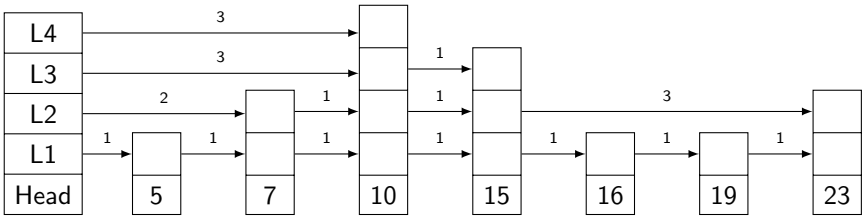

Figure 3.2: Example of a Skip List maintaining distance information between nodes

With these extensions, the Skip List performs very well for common tasks of the Printing Stack. The average complexity of every single Skip List operation never exceeds $\mathcal{O}(\log n)$. For ReactOS, a Skip List has been implemented with 16 pointers per element for managing Printers and Print Jobs. This number of pointers keeps the average algorithm performance for up to $2^{16} = 65536$ elements. A small test suite has been added to guarantee stability and reliability of the abstract data type implementation.

3.7.2 Fast Random Number Generator

As noted beforehand, the implementation of a Skip List depends on a Generator returning geometrically distributed random numbers for $p = 0.5$. In fact, the Generator needs to output an integer between 1 and the array size. It does not need to produce cryptographically secure random numbers though. Even more, the numbers may be fully predictable, because only their distribution is relevant for the Skip List concept to work.

With these given requirements, it was decided to integrate a custom Random Number Generator into the Skip List code instead of depending on an operating system function. Designing a custom Random Number Generator would go far beyond the scope of this thesis. To make a simple Generator available for nonspecialists, Stephen Park and Keith Miller proposed a variant of the Linear Congruential Generator (LCG) in 1988, which is now known as the *Minimal Standard Random Number Generator* [34].

The Skip List's `_GetRandomLevel` function makes use of this Random Number Generator using the revised parameters $a = 48271$ and $n = 2147483647$ [35]. As the returned numbers may be completely predictable, a fixed seed of 1 was defined in the code.

In the beginning, it outputs 31 uniformly distributed random bits. A bitshift to the right is then performed to shift out some bits and only leave one bit for every possible level (as configured through the array size). This method limits the possible Skip List node array size to 31 pointers. Anyway, such a size would be enough to account for up to 2^{31} elements while still having a data structure that keeps the average algorithm complexity. For the Printing Stack, a Skip List with 16 possible levels is used, so half of the random bits are shifted out.

Algorithm 1 Looking up an element and its 1-based index in a Skip List

```
1:  procedure LookupElementSkiplist(Skiplist, Element)
2:      Index ← 1
3:      Node ← Skiplist.Head

4:      for i = Skiplist.MaximumLevel → 1 do
5:          while Node.Next[i].Element < Element do
6:              Index ← Index + Node.Distance[i]
7:              Node ← Node.Next[i]
8:          end while
9:      end for

10:     Node ← Node.Next[0]
11:     if Node.Element ≠ Element then
12:         return ∅
13:     end if

14:     return (Node.Element, Index)
15: end procedure
```

Finally, the uniform distribution is turned into a geometric distribution for $p = 0.5$ by using the *Bit Scan Forward* processor instruction. This instruction counts the bits set to one from the least-significant bit to the most-significant bit and returns the position of the first zero. Hence, a zero in the first bit has a probability of $0.5^1 = 0.5$, a zero in the first two bits has a probability of $0.5^2 = 0.25$, a zero in the first three bits has a probability of $0.5^3 = 0.125$, and so on. As a result, this method delivers a geometric distribution for $p = 0.5$ efficiently.

The full `_GetRandomLevel` listing can be found in Appendix A.1. The function is simple enough to be declared as an inline function.

4 Evaluation

In this chapter, the developed components are evaluated through specific individual tests. These tests do not only cover the actual components making up the Printing Stack, but also verify the correctness of the Skip List implementation.

4.1 Verifying The Random Number Generator

In Section 3.7.2, a fast Random Number Generator has been developed that is designed to return geometrically distributed numbers denoting Skip List levels. The probability parameter of this Generator is defined as $p = 0.5$. Achieving a sufficiently geometric distribution of Skip List levels is crucial for the Skip List performance. The average complexity of the algorithms involved cannot be reached if the Skip List levels follow a different distribution.

To verify the distribution of levels returned by the Random Number Generator, a test application has been written. It calls the Random Number Generator a predefined number of times to simulate the distribution of levels for a Skip List with the same element count. Logarithmic plots of the results of simulations with 1000 and 65536 elements are provided in Figure 4.1.

In both cases, the results show an approximate geometric distribution for the lower levels with some outliers for higher levels. As the number of elements is significantly lower on higher levels, these outliers are negligible. Generally spoken, the more elements are added to a Skip List, the more their assigned levels converge to a geometric distribution. In total, this test shows that the implemented Random Number Generator is feasible to get the desired distribution for the Skip List algorithms.

4.2 Testing The Skip List Implementation

As the Skip List is an integral component of the developed Printing Stack, its algorithms received additional unit testing. In particular, the Skip List for the Printing Stack includes distance arrays for each element to allow a fast lookup of element indexes. This requires the insertion and deletion functions to update both pointers and distances with every operation.

To test all implemented Skip List functions, another test program has been written. It utilizes a Skip List that manages plain integer numbers and sorts them in ascending order. The test first adds 40 random numbers to the list. In a next step,

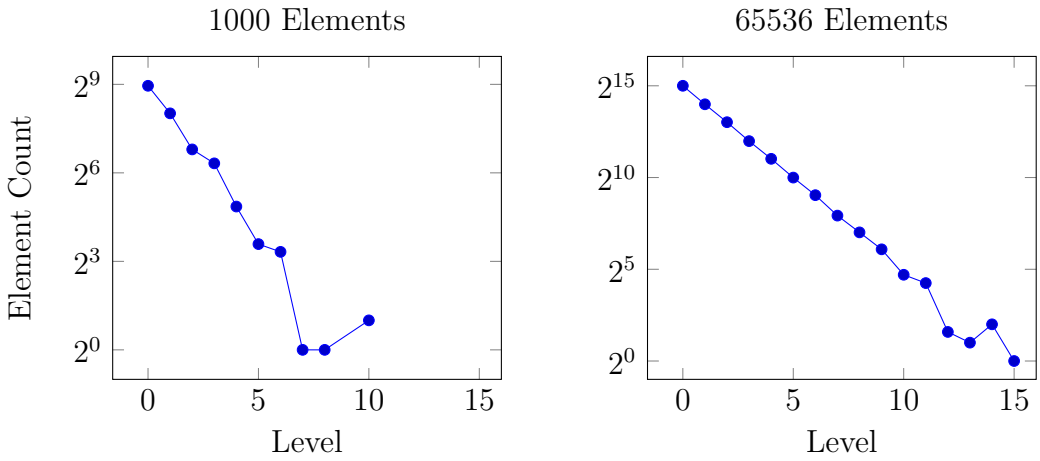

Figure 4.1: Distribution of 1000 and 65536 elements across the levels of a 16-level Skip List using the Minimal Standard Random Number Generator

all numbers in the range 0 to 29 are deleted again. Finally, another batch of 40 random numbers is added to the Skip List. Two lookup operations are performed afterwards, one by index and one by looking for a specific integer. In the end, the structure of the final Skip List is drawn on the screen. The drawing process makes use of both the pointer and distance information. An exemplary output of the test program is depicted in Figure 4.2.

By calling these individual functions in this specific order, all aspects of the Skip List are tested. Typical bugs in abstract data type implementations can be caught, like corrupted structures after an operation, off-by-one mistakes in loops, or similar. The final dump of the resulting structure allows a visual check of the entire Skip List. In case of a bad link to a next element or a wrongly calculated distance, an element would appear misaligned.

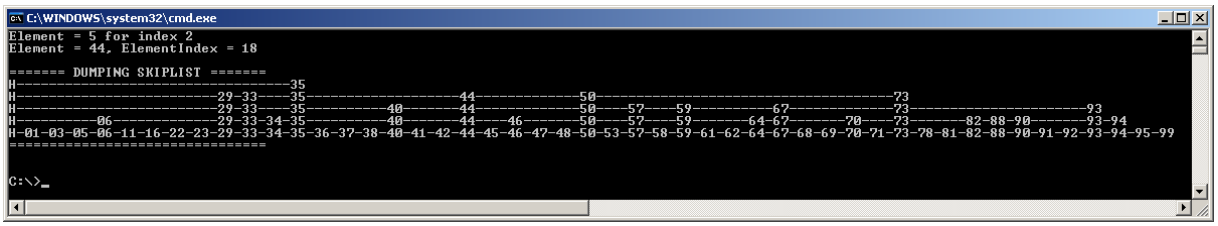

Figure 4.2: Exemplary output of the Skip List test program

4.3 Testing The ReactOS Printing Stack In A Virtual Machine

After individual tests of crucial components, the developed Printing Stack as a whole needed to be tested. This has been accomplished by building an ISO 9660 image file for the entire ReactOS Operating System, including the newly developed Printing Stack components and the *winspool_print* test program. The ReactOS Build System provides the target *bootcd* for this purpose. Calling this CMake target builds all operating system components and finally creates an ISO 9660 image file. Instead of using the default GCC compiler, the operating system has been built using Microsoft Visual C++. This creates the necessary PDB files to enable source-level debugging with WinDbg.

As neither Printer Drivers nor Printer Setup functions exist at this stage of the Printing Stack, Printers cannot be installed through the operating system yet. Without an installed Printer, the Printing Stack cannot be tested though. To circumvent this problem, the ReactOS installation data has been modified to always set up a dummy Printer connected to the first Parallel Port. Without a corresponding Printer Driver, this dummy Printer is still able to forward RAW data to the connected real Printer. As no other datatype nor datatype conversions are supported by now, such a dummy Printer is sufficient for all testing scenarios.

To catch outstanding code bugs early, it has been decided to use a Virtual Machine for testing. This allows to develop and test on the same computer as well as mounting the created ISO 9660 image file in a virtual CD drive instead of burning it to a blank CD. While many different Virtualization products exist today, only some were suitable for evaluating the Printing Stack. This is due to the fact that the Printing Stack outputs data over a Parallel Port. For this purpose, the free VMware Player software turned out to be a viable solution, because it offers a virtual Parallel Port, whose output can be redirected into a file. The returned data can then be examined to validate the correctness of the Printing Stack.

VMware Player has also been configured to emulate a virtual Serial Port and redirect this one to a bidirectional pipe. The WinDbg debugger can then connect to this pipe and act like it was debugging ReactOS on a computer connected through a physical Serial Cable.

By the use of VMware and WinDbg, several bugs have been caught and fixed in a relatively short time. Using a Virtual Machine instead of Real Hardware for these tests has also simplified the deployment of fixes: System components could be exchanged by turning off the Virtual Machine, mounting its virtual Hard Disk on the Host Computer and rebooting ReactOS.

At the end of the testing and fixing phase, the Printing components have finally become robust enough for the *winspool_print* tool. The set of implemented API calls can be used reliably to send RAW Printing data to the Parallel Port. Examination of the redirected Parallel Port output has revealed that a Printer would receive properly formatted data.

4.4 Running The ReactOS Printing Stack On Real Hardware

The final evaluation of the developed Printing Stack happened on a real computer connected to a physical Printer. For this test, a Lenovo ThinkPad X61 with Docking Station has been chosen. This machine's hardware components are known to be supported by ReactOS out of the box without relying on third-party hardware drivers. The Docking Station offers Parallel and Serial Ports not provided by the laptop itself.

Preferably, a Dot-Matrix Printer would have been connected to the Parallel Port of this laptop. Such Printers work character-wise, meaning that they do not await a full page in a Control Language, but output every transmitted character as soon as it arrives. Due to the unavailability of a Dot-Matrix Printer at the time of testing, a Hewlett-Packard DeskJet 710C Inkjet Printer has been chosen instead. This particular Printer expects all incoming data in Hewlett-Packard's proprietary Printing Performance Architecture (PPA) Control Language. The PPA Language is implemented into the included Windows Device Driver, but otherwise largely undocumented. However, Windows Printer Drivers are not usable in ReactOS before the GDI part of the Printing Stack has been implemented. Therefore, another solution was necessary to prepare data in PPA Language.

This solution has been found in the *PNM2PPA* tool. PNM2PPA is a program that converts an input image in Portable Pixmap raster format into appropriate PPA data for supported Hewlett-Packard Printers [28]. It is usually used in conjunction with the GhostScript software, which converts documents in PostScript language into raster formats. Together both applications build a filter chain for CUPS users to let them print on PPA Printers.

As a result of this, a single page document was prepared on a different computer. This document was first converted into PostScript, then processed into a raster image using GhostScript until PNM2PPA finally produced data in PPA format. The created PPA file was then transferred to the laptop running ReactOS. Finally, the *winspool_print* tool read the PPA file and transmitted correctly formatted data to the DeskJet Printer using API functions of the Printing Stack. The Printer reacted accordingly and printed out the previuosly prepared document.

This final step of testing has shown that the implemented features of the Printing Stack can properly communicate with Printers.

5 Conclusion

This thesis presents a Printing Stack for the ReactOS Operating System, which offers compatibility with Windows Printing API functions and extensibility for future additions.

Through extensive research in advance, the official Microsoft documentation for the Windows Printing Stack has been complemented by additional information. An overview of the components involved and their mutual dependencies has been given.

Based on the gathered information, a compatible replacement has been developed for each fundamental component. By the consequent usage of various verification methods during the development process, reliable code and compatibility to Windows Server 2003 is guaranteed. Implementing communication between applications and the Printing Stack consistently over RPC paves the way for a future integration of ReactOS-based Print Servers into a networked environment.

The Skip List abstract data type has been introduced to address the need for a flexible list structure inside Printing Stack components. Its implemented extensions provide performance enhancements when handling large sets of elements.

Final testing of the Printing Stack inside a Virtual Machine and on Real Hardware has demonstrated the functioning of the developed framework.

Within the scope of this bachelor's thesis, only the foundations of the Printing Stack could be implemented. The next logical task would be adding support for native Windows Printer Drivers, which convert EMF data into Printer Control Language. Along with this task comes the implementation of fundamental GDI functions for Printing, e.g. `StartDoc` and `EndDoc`.

In order to provide an intuitive and satisfying user experience, the ReactOS Printing Stack also requires several user interface components to be developed, which guide the user through the installation, configuration, and management of available Printers.

Finally, a modern operating system needs to provide support for a wide variety of Printers out of the box. Due to the vast number of different Printers available, the development focus should lie on Generic Drivers implementing common Printer Control Languages. One example of this is Adobe PostScript.

To put it in a nutshell, Printing is an integral feature of modern desktop operating systems. Providing a suitable Printing Stack for the ReactOS Project enriches the user experience and improves compatibility with many popular applications. All in all, this step makes more usage scenarios imaginable for the Open-Source Windows-compatible operating system.

Appendix

A Listings

This appendix provides the actual C code for some implemented functions discussed in Chapter 3.

A.1 _GetRandomLevel function

The `_GetRandomLevel` function has been introduced in Section 3.7.2. It implements the Random Number Generator required for determining the level of a new element added to a Skip List.

```
// Define SKIPLIST_LEVELS to the maximum number of levels
   the Skip List shall have.
#define SKIPLIST_LEVELS    16

C_ASSERT(SKIPLIST_LEVELS >= 1);
C_ASSERT(SKIPLIST_LEVELS <= 31);

static __inline CHAR
_GetRandomLevel()
{
    // Using a simple fixed seed and the Park-Miller Lehmer
       Minimal Standard Random Number Generator gives an
       acceptable distribution for our "random" levels.
    static DWORD dwRandom = 1;

    DWORD dwLevel = 0;
    DWORD dwShifted;

    // Generate 31 uniformly distributed pseudo-random bits
       using the Park-Miller Lehmer Minimal Standard Random
       Number Generator.
    dwRandom = (DWORD)(((ULONGLONG)dwRandom * 48271UL) %
       2147483647UL);

    // Shift out (31 - SKIPLIST_LEVELS) bits to the right to
       have no more than SKIPLIST_LEVELS bits set.
    dwShifted = dwRandom >> (31 - SKIPLIST_LEVELS);

    // BitScanForward doesn't operate on a zero input value.
```

```
    if (dwShifted)
    {
        // BitScanForward sets dwLevel to the zero-based
           position of the first set bit (from LSB to MSB).
        // This makes dwLevel a geometrically distributed
           value between 0 and SKIPLIST_LEVELS - 1 for p =
           0.5.
        BitScanForward(&dwLevel, dwShifted);
    }

    // dwLevel can't have a value higher than 30 this way,
       so a CHAR is more than enough.
    return (CHAR)dwLevel;
}
```

A.2 _RpcWritePrinter implementation

`_RpcWritePrinter` is one of the RPC server functions implemented in the Spooler Server. It serves as an example how most of the RPC calls are implemented. Basically, Impersonation is performed, the corresponding Spooler Router function called, and finally the security context reverted back to the system context. This process is further illustrated in Figure 2.2.

```
DWORD
_RpcWritePrinter(WINSPOOL_PRINTER_HANDLE hPrinter, BYTE*
   pBuf, DWORD cbBuf, DWORD* pcWritten)
{
    DWORD dwErrorCode;

    dwErrorCode = RpcImpersonateClient(NULL);
    if (dwErrorCode != ERROR_SUCCESS)
    {
        ERR("RpcImpersonateClient failed with error %lu!\n",
           dwErrorCode);
        return dwErrorCode;
    }

    WritePrinter(hPrinter, pBuf, cbBuf, pcWritten);
    dwErrorCode = GetLastError();

    RpcRevertToSelf();
    return dwErrorCode;
}
```

Bibliography

[1] Adobe Systems Incorporated. *Adobe Acrobat X Pro * Set Adobe PDF Properties (Windows)*. 2015. URL: http://help.adobe.com/en_US/acrobat/X/pro/using/WS58a04a822e3e50102bd615109794195ff-7f2a.w.html.

[2] Android Open Source Project. *Android KitKat | Android Developers*. 2015. URL: http://developer.android.com/about/versions/kitkat.html.

[3] Deborah Black. *Microsoft and DCE*. 1993. URL: ftp://ftp.microsoft.com/developr/win32dk/sdk-docs/rpc/DCESIG.PPT.

[4] Vernon Brooks. *IBM PC BIOS source code reconstruction*. 2015. URL: https://sites.google.com/site/pcdosretro/ibmpcbios.

[5] Ross Burton. *gnome-cups-manager - display and edit CUPS printers*. 2003. URL: http://manpages.ubuntu.com/manpages/hardy/man1/gnome-cups-manager.1.html.

[6] Tom Callaway. *The long, sordid tale of Sun RPC, abbreviated somewhat, to protect the guily and the irresponsible*. 2010. URL: http://spot.livejournal.com/315383.html.

[7] Easy Software Products. *CUPS Licensed for Use in Apple Operating Systems!* 2002. URL: https://web.archive.org/web/20020810173413/http://www.cups.org/news.php?V68.

[8] Free Software Foundation. *strings - GNU Binary Utilities*. 2015. URL: https://sourceware.org/binutils/docs/binutils/strings.html.

[9] Dave Gardner. *WINE (WINdows Emulator) Frequently Asked Questions*. 1998. URL: http://www.faqs.org/faqs/windows-emulation/wine-faq/.

[10] Ziliang Guo. *ReactOS Newsletter 54*. 2009. URL: https://reactos.org/newsletter-54.

[11] Alex Ionescu. *The Reactos Project - An Open Source OS Platform for Learning*. 2007. URL: http://mirror.csclub.uwaterloo.ca/csclub/alex-ionescu.pdf.

[12] Ward Rosenberry John Shirley. *Microsoft RPC programming guide*. O'Reilly & Associates, Inc., 1995. ISBN: 1565920708.

[13] Achim Kolacki. *Windows Software Training*. Springer Fachmedien Wiesbaden, 1987. ISBN: 9783528045586.

[14] Laurens Leurs. *The history of PostScript*. 2013. URL: http://www.prepressure.com/postscript/basics/history.

[15] Jean-Baptiste Marchand. *Windows network services internals*. 2006. URL: http://www.hsc.fr/ressources/articles/win_net_srv/index.html.

[16] Microsoft Corporation. *[MS-RPCE]: Remote Procedure Call Protocol Extensions*. 2015. URL: http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5BMS-RPCE%5D.pdf.

[17] Microsoft Corporation. *[MS-RPRN]: Print System Remote Protocol*. 2014. URL: http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5BMS-RPRN%5D.pdf.

[18] Microsoft Developer Network. *Client-Side Rendering*. 2015. URL: https://msdn.microsoft.com/en-us/library/windows/hardware/ff545962%28v=vs.85%29.aspx.

[19] Microsoft Developer Network. *CreateRemoteThread function (Windows)*. 2015. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437%28v=vs.85%29.aspx.

[20] Microsoft Developer Network. *Language Monitors (Windows Drivers)*. 2015. URL: https://msdn.microsoft.com/en-us/library/windows/hardware/ff556450%28v=vs.85%29.aspx.

[21] Microsoft Developer Network. *OpenPrinter function (Windows)*. 2015. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd162751%28v=vs.85%29.aspx.

[22] Microsoft Developer Network. *Selecting a Protocol Sequence*. 2015. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa378665%28v=vs.85%29.aspx.

[23] Microsoft Developer Network. *SetLastError function (Windows)*. 2015. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680627%28v=vs.85%29.aspx.

[24] Microsoft TechNet. *Impersonation*. 2015. URL: https://technet.microsoft.com/en-us/library/cc961980.aspx.

[25] Open Systems Resources. *The NT Insider: Kernel Mode Basics: Splay Trees*. 2008. URL: https://www.osronline.com/article.cfm?article=516.

[26] Kurt Pfeifle. *Dissecting The CUPS Filtering System: A Network Postscript RIP For non-PS Printers*. 2002. URL: http://www.openprinting.org/download/kpfeifle/LinuxKongress2002/Tutorial/V.CUPS-Filtering-Architecture/V.CUPS-Workshop-LinuxKongress2002-Content.html.

[27] Kurt Pfeifle. *KDEPrint*. 2001. URL: http://www.linux-community.de/Internal/Artikel/Print-Artikel/LinuxUser/2001/10/KDEPrint.

[28] PNM2PPA Team. *PNM2PPA GhostScript Print Filter*. 2015. URL: http://pnm2ppa.sourceforge.net/.

[29] Patrick A Powell. *LPRng Reference Manual (For LPRng-3.8.35)*. 2010. URL: http://www.lprng.com/LPRng-Reference/LPRng-Reference.html.

[30] William Pugh. *A Skip List Cookbook*. Tech. rep. Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1990.

[31] William Pugh. *Skip Lists: A Probabilistic Alternative to Balanced Trees*. Tech. rep. Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1990.

[32] ReactOS Team. *ReactOS - New Informations*. 1998. URL: https://web.archive.org/web/19981203081542/http://www.sid-dis.com/reactos/new.htm.

[33] Mario Sixtus. *ReactOS: Das Nicht-Windows*. 2004. URL: http://www.spiegel.de/netzwelt/tech/reactos-das-nicht-windows-a-287199.html.

[34] Keith W. Miller Stephen K. Park. "Random number generators: good ones are hard to find". In: *Communications of the ACM* 31.10 (Oct. 1988), pp. 1192–1201. URL: http://www.firstpr.com.au/dsp/rand31/p1192-park.pdf.

[35] Paul K. Stockmeyer Stephen K. Park Keith W. Miller. "Remarks on Choosing and Implementing Random Number Generators". In: *Communications of the ACM* 36.7 (July 1993), pp. 105–110. URL: http://www.firstpr.com.au/dsp/rand31/p105-crawford.pdf.

[36] Michael Sweet. *Linux Today - A Bright New Future for Printing on Linux*. 1999. URL: http://www.linuxtoday.com/developer/1999060901410NWSM.

[37] Feng Yuan. *Windows Graphics Programming: Win32 GDI and DirectDraw*. Prentice Hall PTR, 2001. ISBN: 978-0130869852.