Transitioning from a.out to ELF

Memories of upgrading NetBSD 1.4.3 to 1.5

Chris Pinnock

20th August 2022

1 Abstract

The original NetBSD release for PC hardware (*i386*) used the *a.out* binary format. At the time, the *a.out* format was approaching its shelf life and other projects were switching to the more modern *Executable and Linking Format* (*ELF*) developed for System V Release 4 and adopted earlier by Linux. The NetBSD 1.5 release [18] transitioned the *i386* and *sparc* ports from *a.out* to *ELF*. Other ports such as *alpha* and *pmax* were introduced to NetBSD with *ELF*. In this paper we describe the binary formats and relive the transition process used by the developers to upgrade a 1.4.3 *a.out* system to a working *ELF*-based system. We will use the QEMU emulator to go back in time to 1999.

2 Disclaimer

Do not attempt to use the images or binaries in any production setting. The images contain software that is over 20 years old and that has security vulnerabilities. Also in this paper, we will be doing most operations using a passwordless root account and this is not recommended in production.

3 Introduction

Why would anyone write a paper about upgrading between such ancient versions of NetBSD? For several platforms, including *i386* and *sparc*, the upgrade from 1.4.3 to 1.5 involved a change of executable file format [18]. The 1.4.3 release used the classical *a.out* format and for selected platforms, the 1.5 release used the more modern *Executable and Linking Format* (*ELF*) format developed for System V Release 4 and used by Linux.

Most users would have upgraded using the NetBSD installer, but at the time I was following the development version of NetBSD between the releases. In 1999, I upgraded using a process designed for following the bleeding edge development release by building from source [19]. This was done on an Intel 486DX2-based PC with 16 megabytes of RAM and 1.2 gigabytes of hard disc [12]. In this paper we will upgrade from 1.4.3 to the last release of the 1.5 branch, 1.5.4.

The order of operations is very important and one wrong step can result in a broken system. The kernel upgrade is relatively straightforward: one builds a 1.5.4 kernel that is capable of executing both *a.out* and *ELF* binaries. At this point, the kernel is still in *a.out* format and the system's boot loader can only boot *a.out* kernels. The crux of the upgrade is building a toolchain of compiler, linker and assembler that are themselves *a.out* but output *ELF* binaries. Once this is done, the libraries and rest of user-land can be built as *ELF* binaries. Finally the kernel can be changed to an *ELF* version and the boot loader can be replaced with one that will boot an *ELF* kernel.

Recently with the help of the QEMU emulator I revisited the upgrade as a nostalgic exercise. There is also plenty to learn from the experience as we will see below. In this paper we will describe the following:

1. Historical aspects of the NetBSD operating system at the time;
2. The object formats *a.out* and *ELF*, their history and differences;
3. An overview of upgrading an *a.out* system to an *ELF* system.

We describe the upgrade process using the sources and you can follow along with the process using the QEMU emulator to upgrade from NetBSD 1.4.3 to 1.5.4. We will assume that you know the basics of the Unix command line. We will also conclude with a brief examination of the *build.sh* system which vastly simplifieds source upgrades.

The files and resources mentioned in the paper can be found here: https://chrispinnock.com/2022/08/20/netbsdaout/

4 NetBSD background

4.1 History

The NetBSD project [20] evolved from the Berkeley Software Distribution (BSD) operating system at the University of California, Berkeley. The history of the system is widely documented elsewhere but it has its roots in the PC port 386BSD and 4.4BSD Lite2 [25]. One of the aims of the NetBSD project is to provide a fully reusable BSD-licensed system [5]. This includes the possibility of forking the code into a private or commercial project.

Another goal of the project is to provide a portable system running on many hardware platforms and by the time NetBSD 1.2 was released, it ran on DEC Alpha (*alpha*), Amiga (*amiga*), Atari (*atari*), HP300 (*hp300*), i386 PCs (*i386*), m68000 Macintosh (*mac68k*), PC532 (*pc532*), DEC MIPS workstations (*pmax*) and Sparc systems (*sparc*) amongst others.

Between the 1.2 [15], 1.3 [16] and 1.4 releases [17], there were major advances including Linux binary emulation, a user friendly installation process for the PC and the NetBSD packages collection which took away the hassle of building third-party software.

The *a.out* format was beginning to show its age and bolting on features to it was becoming difficult. Linux switched from *a.out* to the SVR4 format *ELF* upon the release of Linux 1.2 [1]. The *alpha* and *pmax* ports were officially introduced with *ELF* in NetBSD 1.2 [15].

What remained were the earlier ports such as the *i386* and *sparc*. The NetBSD 1.5 release switched binary format to *ELF* for these platforms. In earlier versions of NetBSD/i386, Linux *ELF* binaries could be executed despite *ELF* not being supported natively.

Despite not support *ELF* natively, Linux *ELF* binaries could already be run under emulation on earlier versions of NetBSD/i386 (see [14] June 1995).

4.2 Navigating the source code

All of the NetBSD source code is available online using the CVS source control system and also viewable at http://cvsweb.netbsd.org/. We will be examining the system using the NetBSD 1.5.4 source code. This can be checked out from CVS or you can download it from my website for convenience:

```
http://downloads.chrispinnock.com/netbsdaoelf/netbsd-1-5.tar.gz
```

On a NetBSD system, the source code usually lives in the */usr/src* directory but it does not have to. From this section onwards, where we refer to a file in the source tree we will give the relative path. So for example on the VM, *bin/Makefile* can be found at */usr/src/bin/Makefile*.

The source code contains a top-level *Makefile* which is referenced by the *make* program to build the source code. It also has instructions for updating using the source code in *UPDATING* and change logs between versions *CHANGES-1-5.{1,2,3,4}*. Here is a breakdown of the source code directories for NetBSD 1.5:

Directory	Purpose
sys	The source code for the kernel
include	The C preprocessor header files containing definitions of functions and system calls
lib	The library source code include the standard C library and math library
bin	Source for statically linked binaries required to bring the system up
sbin	As *bin* but typically requiring super-user (root) privileges
usr.bin	Source for general system-wide binaries, usually dynamically linked
usr.sbin	As *usr.bin* but typically requiring super-user privileges
libexec	Source code for binaries that are not directly run by users (such as Internet daemons)
distrib	Architecture specific tools to build the NetBSD release distribution
etc	Default configuration files and tools to configure a system
share	Files that are shareable between architectures and systems, such as make files, time zone information and manual pages
games	Terminal games including Adventure, Hangman, Tetris and Trek
regress	Regression tests for testing changes to software
dist	External software in a separate directory to ease import into the source tree, such as BIND, DHCP and IPFilter
crypto	Software distributions of cryptographic software including Kerberos, OpenSSL and SSH

Directory	Purpose
gnu	Mostly GNU GPL software including the *gas* (Assembler), GNU *awk*, *egcs* (Compiler), *gdb* (Debugger), *grep* and supporting libraries.

In the early versions of NetBSD, most of the software outside of *gnu* had a four clause BSD-style license [5] which allows reuse provided that copyright notices are displayed, the name of the originator is not used to endorse or promote derived products, and that the software is accepted on an “as-is” basis without warranty.

Most of the software in *gnu* was tools subject to the GNU General Public License (GPL) v2 [11]. GPL software requires the vendor to make source code available if binaries are shipped. This would prevent use of the code by a private commercial product. It was therefore considered important to keep GPL software separately from the rest of the sources and attempt to replace the software with BSD-licensed versions over time.

There is also some non-GPL software in *gnu* such as *sendmail* due to the non-commercial use license. In the latest versions of NetBSD, all external software regardless of license can be found in the *external* directory filed by license-type, replacing most of *dist* and all of the *gnu* directory.

It should be noted that due to the broad range of hardware support of *gcc*, the GNU assembler and compiler software are still the tools of choice in the latest NetBSD releases, but tools such as *grep* and *gawk* have been replaced with other versions.

The cryptographic sources are also kept separately in *crypto* to keep them out of the main source tree. NetBSD 1.5 was the first release to integrate Kerberos, OpenSSL and Secure Shell (SSH) into the operating system. Exporting cryptographic software from the USA was restricted by US law. Such software was treated as munitions until 1992. At the time of the NetBSD 1.5 release, the USA was relaxing its approach to export but some restrictions are still in place [9]. It is possible to set an environment variable to prevent cryptographic software from building and in fact we will use this to simplify our first build below.

On older systems, the BIOS and bootloader could not always boot from a large partition. Usually the system would have a small root partition and a larger */usr* partition. The system libraries would be in */usr*. The binaries in */bin* and */sbin* in the root partition needed to be statically linked so that the system would boot before */usr* was mounted. It also made it easier to recover a damaged system.

On modern systems where it is possible to easier boot by alternate means, from large partitions or in the cloud where systems can be brought up and down very easily, the static linking is no longer a

requirement. More recent versions of NetBSD have switched to dynamically linked binaries in */bin* and */sbin*.

The *regress* directory remains but has been largely replaced by a comprehensive set of tests based on the Automated Test Framework [3].

The BSD-licensing system has been significantly simplified over the years. The interested reader should consult [5].

5 Binary object formats

The purest binary format is simply a file of machine instructions, maybe with some data. The file is loaded into memory and executed directly on the CPU. The DOS *.com* format is an example of such an object format. However, even the earliest home computers had ways of setting up program metadata including a load address and a start execution address.

As years have gone by, things have become more sophisticated. We want to be able to run multiple processes at the same time and we want to be able to load programs into different memory addresses. We want to be able to group common functions into shared libraries that can be updated independently of the other programs. The ability to hold a symbol table in a binary file aids debugging greatly. More recently, object formats have evolved to allow protection of dynamic data segments versus static code. We do not want rogue processes to be able to overwrite code segments with bogus or malicious code.

Typically a binary has a header identifying its type, the machine it is designed to run on, the entry point and other information for the system to run it correctly. On a modern system with shared libraries, the linker *ld* is used to resolve external dependencies in the binary at compile time. The runtime linker *ld.so* is used to “interpret” the binary at runtime and ensure that the relevant shared libraries are loaded into memory. It also ensures that function calls into the libraries are correctly married up with their target locations in memory.

There are various binary formats in use (see [6]) including *a.out*, *COFF* and *ELF*. Originally, *a.out* was superseded by *COFF* in Unix System V which was in turn replaced by *ELF* in System V release 4. We will not describe this format in this paper, because our aim is to migrate a system from *a.out* to *ELF*. We note that the NetBSD/sh3 port (which was incomplete at the time of the 1.5 release) was originally developed using the *COFF* binary format.

We now describe *a.out* and *ELF* at high-level. The interested reader can find a much more substantial treatment in [13]. Also [24] describes *ELF* in the Solaris operating system.

5.1 a.out

The original Unix binary format was baptised *a.out* “a dot out” by Ken Thompson [26]. The name simply stands for *assembler output*. It is usually the assembler that produces an object file. The default filename for the fully linked output of the toolchain is still *a.out* today.

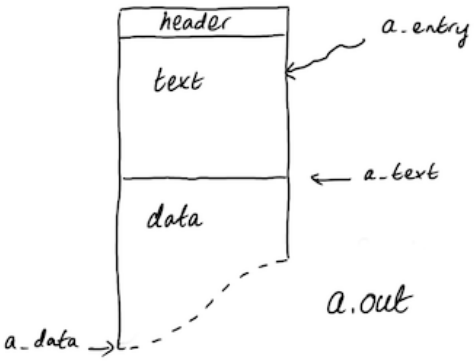


Figure 1: a.out

The *a.out* format was not standardised across Unix platforms and we will reference the NetBSD format [2]. The simplest file consists of a header, executable code (called the text) and initialised data. However, an *a.out* binary file consists of up to seven sections:

Section	Purpose
Header	contains information used to load the file and execute it
Text segment	contains the executable instructions
Data segment	contains initialised data, loaded into writable memory
Text relocations	contains records used by the linker to update pointers in the text when combining binary files
Data relocations	the same, but for the data segment pointers
Symbol table	contains records used by the linker to reference addresses of variables and functions between binary files (such as libraries)
String table	Contains strings corresponding to the symbol names

Seperating the text and data allows the operating system to optimise the use of the binary. For example, it can run two copies of the binary but only hold one copy of the text section in memory

whilst maintaining different data sections for each running copy. The text can also be loaded read-only to prevent tampering during running.

The header is a structure of 8 unsigned long integers (see *sys/exec_aout.h*) as follows:

Field	Purpose
a_midmag	The Machine ID and Magic flags (see below)
a_text	size of the executable code (text)
a_data	size of the initialised data
a_bss	size of the bss segment
a_syms	size of the symbol table
a_entry	the memory address where program execution starts
a_trsize	size of the text relocation section
a_drsize	size of the data relocation section

The bss[1] segment is setup by the kernel when the program is loaded and is a writable area of memory initialised with zeros.

The first number contains the machine ID and magic[2] flags including:

- Flags indicating whether the binary is dynamically linked and containing position independent code (if it is both, it is a shared library);
- The machine ID indicating which machine the binary is intended to run on (e.g. 0x086 for i386 and 0x08a for Sparc);
- The magic number determining the loading convention of the binary.

When storing a number in computer memory that is larger than one byte, there has to be a convention on the ordering of the bytes. Modern processors either order their bytes from the least significant byte to the most significant (6502, ARM, i386) or most significant byte first to the least significant (6809, Sparc, PowerPC). These are called *Little Endian* and *Big Endian*, respectively [3]. On a machine with an i386 processor, the number 0x1234 will be stored as two bytes: 0x34 then 0x12. On the Sparc architecture, it will be stored the other way around: 0x12 and 0x34. Some modern processors can switch Endianness at at initialisation.

[1] There are disagreements about what bss stands for, so we avoid the issue here.
[2] Magic numbers refer to values that identify things such as file types and file systems.
[3] The word Endian comes from Gullivers Travels, where the little Endians liked to crack their eggs at the small end and the big Endians preferred the big end, resulting in a conflict.

The *a_midmag* field is stored in network byte-order which is big Endian. This convention allows all platforms to interpret it and identify binaries intended for other architectures. In network byte ordering, the *a_midmag* field encoding in bits is:

```
FFFFFFmmmmmmmmmmMMMMMMMMMMMMMMMM
```

where F is 6-bit flag number, m is a 10-bit machine ID and M is a 16-bit magic number. Fortunately the operating system provides macros to get the various flags out of the number programmatically so the systems programmer does not have to worry about the underlying architecture of the system.

The magic number refers to the loading conventions used by the operating system. For the *OMAGIC* format, the kernel loads the text and data segments into writable memory. The *NMAGIC* (new magic) format improved the use of virtual memory by *a.out* binaries. The text segment is loaded read-only and the data segment is loaded into the next page boundary after the text.

To simplify memory management, the virtual memory is divided into pages. The pages can be loaded on demand from disc and swapped out when not required. The *ZMAGIC* format introduced alignment of segments to page boundries to simplify paging into memory but at the cost of some disc space. The kernel loads pages into memory from the binary file on demand. Later the *QMAGIC* format achieved the same pageability but with a smaller binary footprint but was discontinued in NetBSD.

The values representing these formats can be found in Octal in *sys/exec_aout.h* and for reference here they are in hexadecimal:

```
Format   Octal   Hex
OMAGIC   0407    0x107
NMAGIC   0410    0x108
ZMAGIC   0413    0x10b
QMAGIC   0314    0x0cc
```

As a worked example, let's look at */bin/sh*. We will examine it from a NetBSD/i386 1.4.3 system. It is in *a.out* format and it is statically linked. We've modified the *hexdump* output [4] here to show the 8 unsigned long integers in the header of the binary.

```
00000000  00 86 01 0b 00 e0 04 00 00 30 00 00 dc 26 00 00
00000010  00 00 00 00 20 10 00 00 00 00 00 00 00 00 00 00
```

Decoding the header on our little Endian i386 system (note the byte ordering):

- The size of the text is 0x4e000 = 319488 bytes
- The size of the initialised data is 0x3000 = 12288 bytes
- The BSS section will be 0x26dc = 9948 bytes
- The symbol table and relocation sections are empty.

[4] A hexdump of this binary can be found here http://downloads.chrispinnock.com/netbsdaoelf/bin-sh-hexdump-143.txt.

- The execution will start at 0x1020.

The *a_midmag* field requires more attention. We ran hexdump on a little Endian system where we see 0x0b018600 and in network byte order this is 0x0086010b.

Exercise: Using bitmaps or using the macros in *sys/exec_aout.h* convince yourself that:

- the flags are all 0 (so not dynamically linked)
- the machine ID is 0x86 (i386)
- the magic number is 0x010b (ZMAGIC)

The binary is statically linked, so no relocation records or symbols are required. Note also that on ZMAGIC, the header is included in the text segment. On the i386 platform a binary is usually loaded at the virtual memory address 0x1000 and hence 0x1020 is the first byte of the executable code after the *a.out* header. The file is 0x51000 = 331776 bytes (319488 + 12288) matching our observations in the header.

There is much more detail than we have covered here, but the general principle is that the binary file not only contains the computer code to run, but also metadata to tell the operating system about its nature.

The *a.out* format has deficiences. For example, there is only one executable segment and modern languages such as C++ need to include initialisation code that is run before the main entry point and code that is run after the main program has exited. This can be seen in NetBSD's Compiler Runtime[5] in *lib/csu* - there are versions for both *a.out* and *ELF*. Although it is possible to get by, more features were needed in the binary format to future-proof it. The *ELF* format is more flexible and we discuss this next.

5.2 ELF

The Executable and Linking Format (*ELF*) was introduced in System V release 4 and consequently Solaris, and has been adopted by Linux and the BSD operating systems. The format is far more flexible and is likely to be used for the foreseeable future.

There are four types of *ELF* file:

1. Relocatable - an object file that still needs to be linked before it can be run
2. Executable - a binary file ready to run except for references to shared libaries that need to be resolved at runtime

[5] The Compiler Runtime is code included in each binary and usually includes some architecture specific code (e.g. to initialise registers).

3. Shared Object - a shared library containing symbol information and runtime code that can be linked at runtime
4. Core file - a file that describes the virtual address space of a process, usually used with a debugger

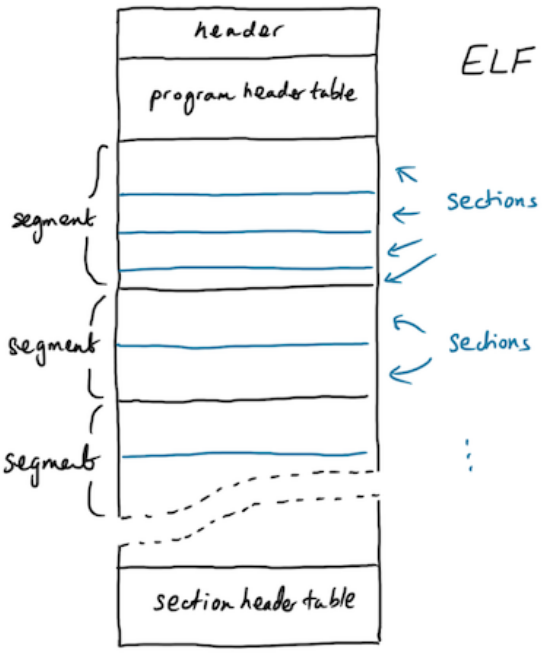


Figure 2: ELF

Every *ELF* file starts with a *header*, which similar to the *a.out* header can be interpreted on machines with different byte orders. The *ELF* file is divided into *segments* which themselves consist of one or more *sections*. The *program header table* describes the segments and the *section header table* describes the sections. A relocatable file will have a program header and an executable file will have a section header table. A shared object will have both.

After the header, comes the program header table (if present), then the sections/segments followed by the section header table (if present).

The *ELF* file can be considered in two ways:

1. The toolchain treats an *ELF* file by considering it as a group of sections which can be processed by the linker;
2. The kernel treats the file by considering its segments which can be loaded into memory.

Thus the program header table is important to the toolchain when combining relocatable files and the section header table is important to the loader when mapping the binary into memory.

The *ELF* header on NetBSD is as follows (for simplicity we describe the 32-bit version):

Field	Use
e_ident	An array of characters used to identify the binary
e_type	Whether the file is Relocatable, Executable, Shared Object or Core
e_machine	Machine type (i386, Sparc, etc)
e_version	The version (usually 1)
e_entry	The entry point of the program if the file is executable
e_phoff	The position of the program header table in the file (or 0 if there isn't one)
e_shoff	The position of the section header table in the file (or 0 if there isn't one)
e_flags	Processor specific flags (out of scope of this paper)
e_ehsize	The size of the *ELF* header
e_phentsize	The size of an entry in the program header table (all are the same size)
e_phnum	Number of program header entries
e_shentsize	The size of an entry in the section header table
e_shnum	Number of section header entries
e_shstrndx	The index of the section containing the section name strings

The *e_ident* is a string array containing the following:

- A string with the ASCII code 0x7f followed by "ELF"
- The address size of the binary (32 or 64-bit)
- The byte order (little or big Endian)
- The *ELF* version (usually 1)
- The Operation system ABI identifier - NetBSD has its own identifier but uses System V's because it does not deviate from the standard
- The ABI version

By using a string of characters, the order is guaranteed on any system. Once the system has read this field, it can make adjustments for the byte order in the rest of the file (although it might not be able to

execute it).

On a recent NetBSD system, *file* will identify an *ELF* binary and library (here on a 64-bit AMD x86-64 system):

```
% file /bin/sh
/bin/sh: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /libexec/ld.elf_so,
for NetBSD 9.99.87, not stripped
% file /lib/libc.so.12.218
/lib/libc.so.12.218: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), static-pie linked, for NetBSD 9.99.87,
not stripped
```

Here we have a 64-bit x86-64 binary using the System V ABI identifier. It is dynamically linked and contains symbols (it is not stripped). The *ld.elf_so* runtime linker is used to "interpret" and load the binary in memory [6]

The *ELF* magic can be seen in a hex dump with the 0x7f byte followed by "ELF".

```
00000000  7f 45 4c 46 02 01 01 00  |.ELF....|
```

The tools available for *ELF* are far kinder to the system programmer than what is available for *a.out*. The *readelf* command outputs information about the file. For example, the header of */bin/sh*:

```
% readelf -h /bin/sh
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4f70
  Start of program headers:          64 (bytes into file)
  Start of section headers:          220136 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         33
  Section header string table index: 31
```

Exercise Use *readelf* to inspect the program headers on an executable file and the section headers on

[6] *ELF* has the facility to supply an interpreters which is usually the runtime linker but does not have to be.

a shared library.

There are various types of sections including:

1. PROGBITS - program “bits” such as executable code, data and debugging symbols
2. SYMTAB - a symbol table containing the symbols needed for the linker
3. DYNTAB - a symbol table containing symbols needed for the dynamic linker
4. STRTAB - a string table (similar to the *a.out* string table)

Sections also have flags including:

1. ALLOC - the section is memory resident
2. WRITE - the section is writable in memory
3. EXECINSTR - the section contains machine code

Example sections include:

- **.text** - a section of type PROGBITS with flags ALLOC and EXECINSTR, the direct analogue of the *a.out* text section.
- **.data** - a section of type PROGBITS with flags ALLOC and WRITE, similar to the *a.out* data section.
- **.rodata** - similar to **.data** but without WRITE, so used for read-only data
- **.init** - PROGBITS with ALLOC and EXECINSTR - similar to **.text**, but this contains code that is executed before the main program starts
- **.fini** - similar to **.init**, but contains code that is executed after the main program exits

The ability to include sections such as **.init** and **.fini** make *ELF* more suitable for modern languages that need to do significant initialisation before the actual code runs. These sections are used by the *ELF* version of the NetBSD Compile Runtime.

In a relocatable file and a shared object, the section header table holds details of the sections including their types, sizes and flags.

In an executable file and a shared object, the program header table holds details of the segments in much the same way, although the information is designed for the kernel to be able to map the program into memory and execute it.

There is much more detail that we have presented here and the interested reader should consult the NetBSD *ELF* manual page [8] and Linkers and Loaders [13] 3.7.

6 Recreating the experience

6.1 Overview

As we outlined in the introduction, the approach to upgrading is as follows:

1. Setup our environment - either using a prepared QEMU image or installing one from a NetBSD 1.4.3 ISO (or for the brave a real-life *i386* machine);
2. Backup the *a.out* libraries;
3. Build a new 1.5.4 kernel enabled for *ELF* binaries but with a compatibility layer for *a.out*;
4. Setup the environment to build 1.5.4;
5. Build a toolchain that is *a.out* itself but outputs *ELF* binaries;
6. Rebuild the includes files and key libraries in *ELF* format;
7. Rebuild the user-land;
8. Build an *ELF* kernel and new boot loader.

The method that this paper is based on was written by Christos Zoulas in 1999 [19] with work undertaken by the NetBSD development team, notably Paul Kranenburg who worked on the NetBSD/sparc port.

6.2 Setting up QEMU

QEMU emulates many system architectures including *i386* architecture. You can get prebuilt QEMU binaries on most platforms. On the BSD operating systems, QEMU is available in the packages or ports collections, on macOS, QEMU is available in *homebrew* and on Debian/Ubuntu Linux it can be installed with *apt*.

```
# NetBSD from source
cd /usr/pkgsrc/emulators/qemu && make install
# or using pkgin
pkgin install qemu

# OpenBSD
pkg_add qemu

# FreeBSD
pkg install qemu

# Debian/Ubuntu Linux
apt install qemu

# macOS (with Homebrew, https://brew.sh)
brew install qemu
```

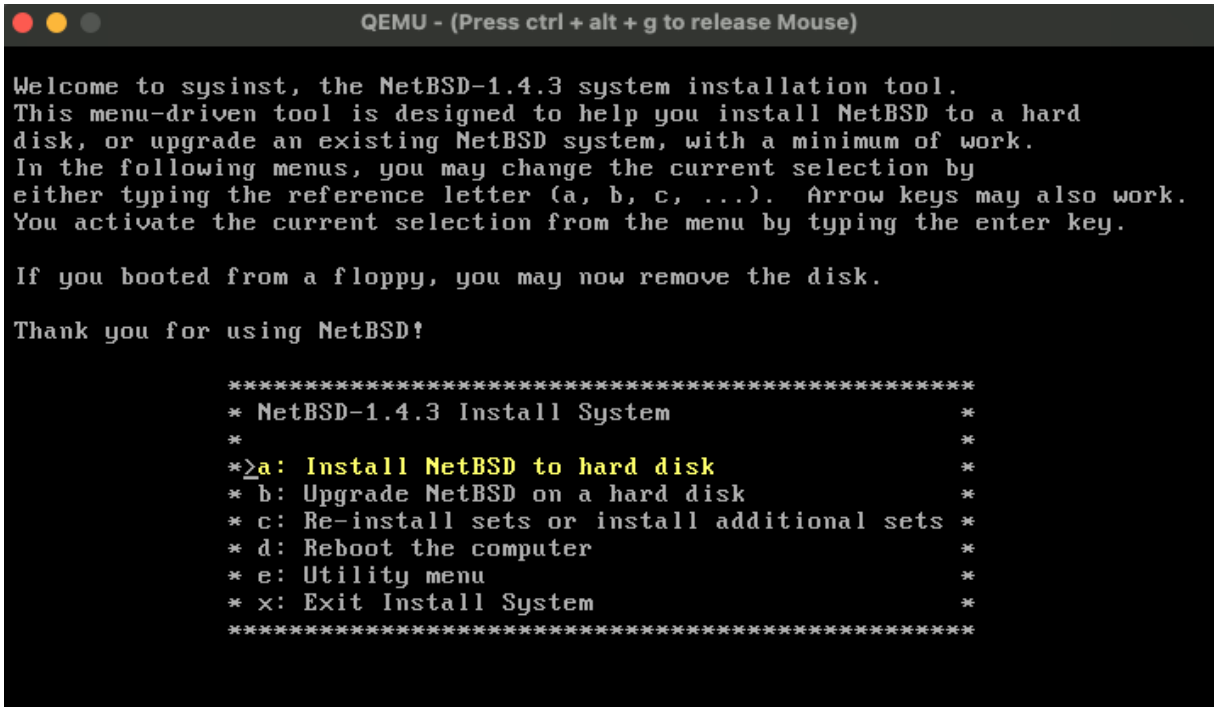
From now on where you see **%** in the commands text, it will mean a command line prompt on the host machine running QEMU and usually **#** will refer to a command line prompt on the QEMU emulator. In quoted shell code, **#** will refer to a comment.

I have provided an image of NetBSD 1.4.3 with the sources ready for 1.5.4. To get the image use your favourite web browser, *ftp -a*, *wget* or *curl*. Once downloaded, decompress the file with *bunzip2* and run QEMU.

```
% wget http://downloads.chrispinnock.com/netbsdaoelf/netbsd-disk-i386-143.
    img.bz2
% cp netbsd-disk-i386-143.img.bz2 netbsd-disk-i386-143.img.bz2.backup
% bunzip2 netbsd-disk-i386-143.img.bz2
% qemu-system-i386 -hda netbsd-disk-i386-143.img \
    -net user -net nic,model=ne2k_pci -boot c -display curses
```

Not all QEMU installations support the *-display curses* option. You can remove it from the command line and QEMU will use a graphics window instead of the terminal. However, it will be easier to copy and paste strings into the terminal if you use curses.

6.3 Installation from ISO



Alternatively I have provided an ISO image of 1.4.3 should you want to install the operating system from scratch yourself.

```
% wget http://downloads.chrispinnock.com/netbsdaoelf/NetBSD143.iso.bz2
% bunzip2 NetBSD143.iso.bz2
% qemu-img create -f qcow2 netbsd-disk-i386.img 20G
% qemu-system-i386 -hda netbsd-disk-i386.img \
    -net user -net nic,model=ne2k_pci -cdrom NetBSD143.iso \
    -boot d -display curses
```

If you choose to install the image yourself, you will need to get a copy of the NetBSD 1.5.4 sources. I have provided these at the same site and they can be decompressed to */usr/src*. For example, from the installed VM:

```
% cd /usr
% ftp -a http://downloads.chrispinnock.com/netbsdaoelf/netbsd-1-5.tar.gz
% tar zxf netbsd-1-5.tar.gz
```

6.4 Orientating the virtual machine

Once booted, you will be about to log in as *root* with an empty password. Welcome to 1999. The consumer PC hardware was 32-bit and NetBSD supported the 386DX up to the 686-class Pentium II. There was no *ssh* or *cvs* in the default installation. The shell was *csh* by default for root. But the system was complete in the sense that it could build itself, provide a complete development environment and a complete workstation environment with X windows.

Furthermore, many software packages were available in the NetBSD packages collection [21] and Linux software could be run under emulation. In particularly, the Netscape browser worked very well.

The virtual machine image provided has some configuration and source ready to speed up the process:

1. The source code for 1.5.4 can be found in */usr/src*.
2. The virtual machine has been setup to use DHCP and get an IP address. The *dhclient* process will try to get addresses on all interfaces. It can either be reconfigured (exercise for the reader with *rc.conf*) or safely killed if you don't need network access.
3. *csh* can be a little cumbersome to use and it shows its age. I have compiled an old version of *tcsh* compatible with this version of NetBSD. You can download it from my website. If network connectivity is working on your virtual machine, this should work:

```
# ftp -a http://downloads.chrispinnock.com/netbsdaoelf/tcsh-netbsd-143-
   aout.gz
# gunzip tcsh-netbsd-143-aout.gz
# chmod +x tcsh-netbsd-143-aout
# ./tcsh-netbsd-143-aout
```

6.5 Backups and preserving the *a.out* libraries

If you were doing this on a live production system, you would take copious backups at this point. The beauty of recreating the upgrade with an emulator is that you can start again from the initial image should something go wrong.

However we do need to preserve the *a.out* versions of the system libraries. The libraries are common routines and functions used by other programs. The most important one is the standard C library.

We mentioned in the introduction that NetBSD introduced Linux emulation in 1.2. The Linux shared libraries are placed in */emul/linux* and are used in preference to the regular libraries when executing a Linux binary.

In the next step, we will add an option to the kernel, COMPAT_AOUT, which will enable it to run *a.out* binaries even when the system is setup for *ELF* binaries by referencing another directory */emul/a.out* for the *a.out* libraries.

We copy the existing *a.out* libraries into */emul/aout*. Here we have included the X11 libraries as well, but they are not essential for our simulation.

```
# mkdir -p /emul/aout/usr/lib /emul/aout/usr/X11R6/lib
# cp -p /usr/lib/*.so* /emul/aout/usr/lib/
# cp -p /usr/X11R6/lib/*.so* /emul/aout/usr/X11R6/lib/
```

When the kernel runs an *a.out* binary, it will use the file hierarchy in */emul/aout* before using the files in the regular directories. It's vital we do this because at one point of the upgrade we will update the libraries in */usr/lib* with *ELF* versions.

At any stage of the process, if you want to make a backup, the easiest way to do it is to *halt* the NetBSD virtual machine, make a copy of the hard disc image and restart it.

6.6 Building a new kernel

Now you have a working NetBSD/i386 1.4.3 system with the sources for 1.5.4 in */usr/src*. It's time to build a new kernel. In order to do this we will use the *config* command to setup the build, but we will need to update *config* first so that it understands all the options in our kernel configuration file.

```
# cd /usr/src/usr.sbin/config
# make && make install
```

Now to make our new kernel. We head into the system area of the sources and edit the GENERIC configuration file to add some options. The GENERIC kernel configuration is designed to run on most hardware available. We could trim down the configuration to just the things we need, but we leave this as an exercise for the reader.

We need to make sure that the following options are in the configuration file. When the team wrote the *ELF* FAQ [19] over 20 years ago, it was written with the development version in mind. Strictly we should not have to add these options - they will be somewhere in the kernel build files, but to be sure let's add them and understand what they will do.

```
options EXEC_ELF
options COMPAT_AOUT
```

You can add them near to the Compatibility Options section in the GENERIC file but order typically does not matter. The first option includes kernel support to execute the *ELF* binaries. The second provides options to execute the *a.out* binary format but also to help the linker transparently find the *a.out* libraries in the emulation directories */emul/aout* we made earlier.

We will use *ed* [7] to put the two lines in the middle of the file. You can use the editor of your choice.

```
# cd /usr/src/sys/arch/i386/conf
# ed GENERIC
71p
a
options EXEC_ELF
options COMPAT_AOUT
.
wq
```

Once this is done, you can configure and build the kernel.

```
# config GENERIC
# cd ../compile/GENERIC
# make depend && make
```

The *make depend* step ensures that a dependency tree is setup that *make* can use to correctly build all the necessary source files and drivers included in the supplied configuration.

Once the kernel is built, we preserve the current 1.4.3 kernel, copy the new one into place and reboot. I have also kept a second copy of the 1.5.4 kernel.

```
# cp /netbsd /netbsd.143
# cp netbsd /netbsd.154.aout
# cp netbsd /netbsd
# shutdown -r now
```

Building a kernel for the latest NetBSD version can be done with exactly the same process today (or, of course, one can also use *build.sh* which we will describe later).

6.7 Setting up to build user-land

The system should have rebooted and you can log in as root with an empty password. If you prefer to use *tcsh*, please run it again now. You will observe that some things in user-land are not working as they should. For example, *ps* cannot converse with new kernel properly:

```
# ps aux
ps: proc size mismatch (10192 total, 680 chunks)
ps: statfs on /proc failed: No such file or directory
ps: fallback on /proc-based lookup also failed.  Giving up...
```

This will be fixed later when we rebuild *ps* and the rest of user-land.

The tool that builds the software on NetBSD is *make*. The supporting Makefiles in NetBSD 1.5 changed significantly from 1.4.3 and the newer version of *make* is needed to understand them properly.

First we rebuild *make* and install it. We also clean the directory afterwards because later on we want to build an *ELF* version of it and it is better that there are no object files around.

```
# cd /usr/src/usr.bin/make
# make && make install && make clean
```

Then install the new system makefiles. These are used in the source files to avoid repeating common routines and set default targets such as “build”, “install” and others.

```
# cd /usr/src/share/mk && make install
```

6.8 Build variables

If you do not want to understand details about the make variables we will use, you can safely skip this section.

The NetBSD system has a set of default Makefiles in */usr/share/mk*. These files contain common routines to make life easier for the developer. For example, *bin/Makefile* is incredibly simple to specify by including the common file *bsd.subdir.mk* and setting the SUBDIR variable to the directories that need to be traversed to build the binaries.

```
#   $NetBSD: Makefile,v 1.18 1999/11/23 05:28:15 mrg Exp $
#   @(#)Makefile    8.1 (Berkeley) 5/31/93

SUBDIR=cat chio chmod cp csh date dd df domainname echo ed expr hostname \
    kill ksh ln ls mkdir mt mv pax ps pwd rcp rcmd rm rmdir sh \
    sleep stty sync test

.include <bsd.subdir.mk>
```

Similar *bin/cat/Makefile* for the program *cat* contains just enough that when including *bsd.prog.mk*, make will build the *cat* binary and manual pages. We will see what the WARNS variable does later.

```
#   $NetBSD: Makefile,v 1.9 1999/07/08 01:56:09 christos Exp $
#   @(#)Makefile    8.1 (Berkeley) 5/31/93

WARNS=2
PROG=   cat

.include <bsd.prog.mk>
```

Throughout the course of the rebuild, we will set environment variables to change the way in which the binaries are built. For example, we will initially set OBJECT_FMT to *a.out* and *make* will use this variable.

```
% setenv OBJECT_FMT a.out
```

We will assume *csh/tcsh* syntax throughout the paper and use *setenv*. If you have decided to use *sh*, *ksh* or similar, you will need to do something different:

```
$ OBJECT_FMT=a.out
$ export OBJECT_FMT
```

The variables we will set are as follows, mostly in the environment but sometimes directly to *make*.

- OBJECT_FMT
- BOOTSTRAP_ELF
- MKCRYPTO
- NOGCCERROR
- NOLINT
- WARNS

All *make* variables pertinent to the build are documented in *share/mk/bsd.README*. In the previous section, we have installed the 1.5.4 versions on our system. Let's look through these files to see their impact.

1. OBJECT_FMT

This *make* variable is used to control the binary format outputted by the compiler toolchain. By default, the 1.5 Makefile support files set this variable to "ELF" for ports including *alpha*, *i386*, *pmax*, *powerpc*, and *sparc*. For the *sh3* port, it would be set to "COFF". For every other port, it was set to "a.out".

We will use this variable in the first instance (together with BOOTSTRAP_ELF) to build a compiler toolchain from the 1.5.4 sources that are *a.out* binaries themselves but will output *ELF* binaries when run.

Exercise: Using *grep*, search the 1.5.4 sources for OBJECT_FMT and see if you can see what the outcomes on the *i386* platform are when it is set to "a.out" or "ELF".

For example, in *libexec/ld.elf_so/Makefile* (the Makefile of the *ELF* loader), the vast majority of the file is ignored if OBJECT_FMT is not set to "ELF" and so the *ld.elf_so* would not be build when OBJECT_FMT is set to "a.out".

As you would expect, the variable appears in the Makefiles for the compiler and assembler (amongst others) because it will affect how they are built. Similarly the correct Compiler Runtime will be included in the outputted binary.

You do not have to understand all of the implications here before moving on. The key takeaway is that the variable controls the outputted binary format.

2. BOOTSTRAP_ELF

This variable can be found in the following programs and libraries, namely the toolchains:

- *gdb* - the GNU Debugger
- *gas* - the GNU Assembler
- *egcs* - the GNU Compiler [7]
- *ld* - the GNU linker
- *libbfd* - the Object File library, which provides an abstraction to the object file library regardless of format.

```
% grep -rl BOOTSTRAP_ELF *
gnu/usr.bin/gdb/Makefile
gnu/usr.bin/gas.new/arch/m68k/Makefile.inc
gnu/usr.bin/gas.new/arch/i386/Makefile.inc
gnu/usr.bin/gas.new/arch/sparc/Makefile.inc
gnu/usr.bin/egcs/Makefile.inc
gnu/usr.bin/ld.new/Makefile
gnu/lib/libbfd/Makefile
```

The assembler and linker have *.new* in their directory names because it is sometimes necessary to have two versions of the programs in the NetBSD source tree whilst migrating between them.

The purpose of the variable is to change the development environment to create a toolchain that outputs *ELF* binaries instead of *a.out* binaries, that is, to bootstrap an *ELF* environment.

We will not need to build the debugger (*gdb*) as part of our bootstrap process, but it would have been important to the developers at the time when working on the migration between *a.out* to *ELF*.

Let's look at an example from above to see what is happening. From *gnu/usr.bin/gas.new/arch/i386/Makefile.inc*, *-DDEFAULT_ELF* is being added to the preprocessor flags. Within the *gas* sources, DEFAULT_ELF is

[7] At the time the GCC project had forked into two projects, gcc and egcs. They eventually merged back into one.

used as a preprocessor variable to include different code segments that will result in the assembler outputting *ELF* files.

```
.if (${OBJECT_FMT} == "ELF") || defined(BOOTSTRAP_ELF)
OBJ_FORMAT_C=   obj-elf.c
CPPFLAGS+=-DDEFAULT_ELF
.else
OBJ_FORMAT_C=   obj-aout.c
.endif
```

Similarly the *egcs* Makefile adds -DDEFAULT_ELF to the preprocessor flags. If you want more detail here, please look at the sources in the *gnu* directory.

3. MKCRYPTO

If this variable is set to “no”, certain programs and libraries containing cryptographic code will not be built. We discussed the implications of cryptographic code earlier. However it may be desirable to remove such software if the target system is constrained on CPU and memory.

One can see this in *lib/Makefile*, but in other places too. The library subdirectories below are skipped if MKCRYPTO is set to no.

```
.if (${MKCRYPTO} != "no")
# OpenSSL libraries.  NOTE!  WE DO NOT TRAVERSE INTO libdes FOR A REASON!
SUBDIR+= libcrypto libssl

:# Heimdal Kerberos 5 libraries
SUBDIR+= libroken libvers libcom_err libsl libss libasn1 libkrb5 libhdb \
         libkadm5srv libkadm5clnt libgssapi

# KTH Kerberos 4 libraries
SUBDIR+= libkrb libkdb libkadm libkafs
SUBDIR+= libkstream

.if (${MKCRYPTO_IDEA} != "no")
SUBDIR+= libcrypto_idea
.endif  # MKCRYPTO_IDEA != no

.if (${MKCRYPTO_MDC2} != "no")
SUBDIR+= libcrypto_mdc2
.endif  # MKCRYPTO_MDC2 != no

.if (${MKCRYPTO_RC5} != "no")
SUBDIR+= libcrypto_rc5
.endif  # MKCRYPTO_RC5 != no

.endif  # MKCRYPTO != no
```

We will set MKCRYPTO to no on the first build to reduce the complexity of the build.

5. NOLINT

The *lint* program is used to check code for common coding mistakes, bugs, portability errors and use of best practices. The build system will normally attempt to build "lint libraries" for library functions and these are used to check programs for compatibility by *lint*. We will turn this off to avoid initial build problems by setting NOLINT.

6. WARNS and NOGCCERROR

If the WARNS variable is defined, the common routines from *bsd.sys.mk* will set extra compiler flags so that the compiler warns on certain bad programming practices. In most places (see for example *bin/Makefile.inc*), WARNS=1, but some programs can build with WARNS=2 (e.g. *bin/cat* as we saw above). Here is the *Makefile* code:

```
.if defined(WARNS)
.if ${WARNS} > 0
CFLAGS+= -Wall -Wstrict-prototypes -Wmissing-prototypes -Wpointer-arith
# XXX Delete -Wuninitialized by default for now -- the compiler doesn't
# XXX always get it right.
CFLAGS+= -Wno-uninitialized
.endif
.if ${WARNS} > 1
CFLAGS+=-Wreturn-type -Wcast-qual -Wpointer-arith -Wwrite-strings
CFLAGS+=-Wswitch -Wshadow
.endif
.endif
```

If NOGCCERROR is set, the compiler will just warn and not exit. The default behaviour is to exit with an error by adding the compiler flag *-Werror*.

```
.if !defined(NOGCCERROR)
CFLAGS+= -Werror
.endif
```

Please refer to the GCC [10] manual for documentation on the warning types. One example *-Wmissing-prototypes* will cause the compiler to complain if a function is not correctly defined. In one place during the build we will set WARNS=0 to avoid the build stopping.

It should not be necessary to set WARNS, NOGCCERROR and NOLINT normally. By setting them, we will remove safeguards from the build process. We will take it on faith that the code is correct or good enough to build to get us to the next stage.

6.9 Building the toolchain

Now we get onto the hardest bit of the upgrade. We must construct a compiler, assembler and linker that are *a.out* format but output *ELF* binaries. We need to build them and **not** install them individually, but install them at the same time so that they work together properly.

As we have described above, the 1.5.4 Makefiles support the transition. To ensure that the desired toolchain is built with the *a.out* object format, set the following environment variables[8] :

```
# setenv OBJECT_FMT a.out
# setenv BOOTSTRAP_ELF yes
```

Now bootstrap the toolchain in this order. First we build the Object File library (*libbfd*), then we build the binary utilities (*binutils*) for manipulating object files. Then we build the assembler (*gas.new*), the linker (*ld.new*) and finally the compiler (*egcs*).

Do not accidentally *make install* at this stage. The tools are *a.out* binaries that output and process *ELF* binaries. Therefore the tools need to be installed at the same time for them to work together properly. If one of them is installed before the others are ready, they will not work together correctly.

```
# cd /usr/src/gnu/lib/libbfd && make
# cd /usr/src/gnu/usr.bin/binutils && make
# cd ../gas.new && make
# cd ../ld.new && make
# cd ../egcs && make
```

Now we can install all of the tools above. We will also clean the directories, because later on we will want to build *ELF* versions of them. Do not install the *libbfd* library - the other binaries use what has been built in the *libbfd* directory.

```
# cd /usr/src/gnu/usr.bin/binutils && make install
# cd ../gas.new && make install
# cd ../ld.new && make install
# cd ../egcs && make install
```

We have crossed the first hurdle. The next step is to start building *ELF* libraries and binaries, but first we need to clean up to remove any *a.out* objects in the source tree. For example, the tools we just installed will have *a.out* objects and we want to rebuild them as *ELF* binaries in the next step.

```
# cd /usr/src && make cleandir
```

[8] The original *ELF* FAQ [19] suggests setting DESTDIR to /../. to ensure that the resultant binaries are installed in the correct root directory and not the emulation directories /emul but I have not found this necessary with the sources used in this paper.

6.10 Rebuilding the key libraries

Now we need to tell our Makefiles we intend to output *ELF* files. Also we may will encounter some minor errors and source code checking, so we will ignore any warnings from the compiler. NetBSD 1.5 and later included libraries and tools with cryptographic software. On the first build, we can ignore these tools to simplify and speed up the build. We set three environment variables to instruct *make* to behave to our demands.

```
# setenv OBJECT_FMT ELF
# setenv NOGCCERROR
# setenv MKCRYPTO no
```

Before we start to build the libraries and binaries, we need to install the latest includes files. These are the .h files used by the sources for definitions of functions and structures.

```
# cd /usr/src
# make includes
```

We need to build the Compiler Runtime support before anything else. Part of the Compiler Runtime is specific to the target architecture and is written in assembly language. Essentially it provides supporting code to initialise and exit programs. We need to build up to date *ELF* compatible versions of these objects and they will use the **.init** and **.fini** sections we described earlier.

```
# cd /usr/src/lib/csu
# make
# make install
```

We now make and install *ELF* versions of all of the standard NetBSD libraries. This includes in the standard C library, the standard mathematics library and so on. We set the *make* variable NOLINT to 1 here because we want to avoid checking the code for errors with *lint* - we'll take it on good faith that the sources work.

```
# cd /usr/src/lib
# make NOLINT=1
# make NOLINT=1 install
```

The next step is to build and install the *ELF* runtime linker. This tool is responsible for loading the necessary shared libraries into memory when executing an *ELF* binary.

```
# cd /usr/src/libexec/ld.elf_so
# make
# make install
```

And finally we build *ELF* versions of the GPL licensed libraries.

```
# cd /usr/src/gnu/lib
```

```
# make
# make install
```

Now we have a system with *ELF* libraries ready to be linked with freshly built *ELF* binaries.

6.11 Finalising userland

To make life easier, we will upgrade two tools that are commonly used in the build. The first is *lint* which is used to check source files for potential errors. The second is *yacc* which is used to build C source code files from language grammar files. By rebuilding these now, we will avoid warnings and build errors[9].

```
# cd /usr/src/usr.bin/xlint && make && make install
# cd ../yacc && make && make install
```

Now we can complete our installation of everything that is left, using objects we have compiled already. This part of the build takes the longest time and again we ask the compiler to continue even after a warning.

```
# cd /usr/src && make WARNS=0 && make install
```

Once this is finished, the system will be in *ELF* format, except for the cryptographic sources. For example, the Bourne shell is now built and ready to run in *ELF* format, statically linked, the *vi* editor is built dynamically linked and the C library is built as an *ELF* shared object.

```
# file /bin/sh /usr/bin/vi /usr/lib/libc.so.12.62.1
/bin/sh:                  ELF 32-bit LSB executable, Intel 80386, version
   1, statically linked, stripped
/usr/bin/vi:              ELF 32-bit LSB executable, Intel 80386, version
   1, dynamically linked (uses shared libs), stripped
/usr/lib/libc.so.12.62.1: ELF 32-bit LSB shared object, Intel 80386,
   version 1, not stripped
```

Finally we can make the whole system again without the relaxation of compiler warnings and including the cryptographic sources. The *build* target builds and installs everything in the right place. It is safest to clean up older objects just in case.

```
# unsetenv MKCRYPTO
# make cleandir
# make build
```

[9] specifically, there are lots of warnings/errors if *lint* is not up to date and errors when building *ftpd* if *yacc* is not up to date. These errors may interrupt the build.

6.12 Finishing with an ELF kernel

We have built libraries and binaries in *ELF* format and the only thing left to address is the kernel, which is still an *a.out* binary. Before we update the kernel, we must update the boot blocks to a version that understands how to load an *ELF* file otherwise we will not be able to boot the system. Let's tackle this first.

The directory */usr/mdec* contains machine dependent files including the boot loaders. The recent *make build* process will have installed the latest versions [10].

```
# cd /usr/mdec
# ./installboot ./biosboot.sym /dev/rwd0a
```

Now back to the new kernel. First we clean our compile directory to remove any *a.out* objects, reconfigure and rebuild:

```
# rm -rf /usr/src/sys/arch/i386/compile/GENERIC
# cd /usr/src/sys/arch/i386/conf
# config GENERIC
# cd ../compile/GENERIC
# make depend && make
```

Once built, we preserve a copy of the new kernel, install it and reboot.

```
# cp netbsd /netbsd.154.elf
# cp netbsd /netbsd
# shutdown -r now
```

The system should reboot properly and we have finished our upgrade, other than perhaps tidying up a few files in */etc*.

```
# file /netbsd*
/netbsd:          ELF 32-bit LSB executable, Intel 80386, version 1,
    statically linked, not stripped
/netbsd.143:      NetBSD/i386 demand paged executable not stripped
/netbsd.154.aout: NetBSD/i386 demand paged executable not stripped
/netbsd.154.elf:  ELF 32-bit LSB executable, Intel 80386, version 1,
    statically linked, not stripped
```

If we were doing this for real, we would now go and recompile all of our third-party software in the new binary format. We don't need to do this in our example and we can safely remove the */emul/aout* directory.

[10] NetBSD 1.5 was the first release to ship with a boot block *biosboot_com0.sym* used to boot on a serial console. This is useful for headless servers and also useful in virtual machine environments such as QEMU.

7 Exercise

Now that we have completed the upgrade from 1.4.3 to 1.5.4 on NetBSD/i386, we can perform a similar operation on NetBSD/sparc. I have provided an ISO image of the NetBSD/sparc 1.4.3 installer and this will boot on the QEMU Sparc emulator.[11]

```
% wget http://downloads.chrispinnock.com/netbsdaoelf/NetBSD143-sparc.iso.
   bz2
% bunzip2 NetBSD143-sparc.iso.bz2
% qemu-img create -f qcow2 netbsd-disk-sparc.img 20G
% qemu-system-sparc -hda netbsd-disk-sparc.img \
    -net user -net nic -cdrom NetBSD143-sparc.iso -boot d -nographic
```

Once installed, you can shutdown the VM and restart the VM:

```
% qemu-system-sparc -hda netbsd-disk-sparc.img \
    -net user -net nic -boot c -nographic
```

Then on the VM download the sources and proceed with the upgrade (using *tcsh* is optional):

```
# dhclient
# ftp -a http://downloads.chrispinnock.com/netbsdaoelf/tcsh-netbsd-143-
   sparc-aout.gz
# gunzip tcsh-netbsd-143-sparc-aout.gz
# chmod +x tcsh-netbsd-143-sparc-aout
# ./tcsh-netbsd-143-sparc-aout
# cd /usr
# ftp -a http://downloads.chrispinnock.com/netbsdaoelf/netbsd-1-5.tar.gz
# tar zxf netbsd-1-5.tar.gz
```

To edit files, you will need to set the term variable appropriately. It's likely that you are working on an *xterm* compatible terminal, so in *csh* and *tcsh* you can use:

```
# set term = xterm
```

Remember you are building for the *sparc* platform, not *i386*, so the system files will be in *sys/arch/sparc*.

When building the kernel, you will need to workaround some errors. You can work around them as follows - this is not particularly elegant, but will get us to where we need to be. Some hints:

1. Temporarily remove or comment out the *memcpy* definitions from *sys/lib/libkern/libkern.h* to avoid conflicts with existing definitions:

```
   167  int      memcmp __P((const void *, const void *, size_t));
```

[11] At the time of writing, under the QEMU 9.2 Sparc emulator on Mac, NetBSD/sparc 1.4.3 crashes when using a graphic console (i.e. without *-nographic*).

```
   168  void    *memcpy __P((void *, const void *, size_t));
   170  void    *memset __P((void *, int, size_t));
```

2. Remove bpf and dependencies from the kernel (you can live without these temporarily). Remove these lines (assuming you have not edited GENERIC):

```
   183  options         PPP_FILTER      # Add active filters for ppp (via
      bpf)
   488  pseudo-device   bpfilter        8
   500  pseudo-device   vlan
```

Both these can be achieved as follows:

```
# cd /usr/src/sys/lib/libkern/
# cp -p libkern.h libkern.h.orig
# ed libkern.h
170d
167,168d
wq
# cd /usr/src/sys/arch/sparc/conf
# cp GENERIC GENERIC.orig
# ed GENERIC
183d
488d
500d
wq
```

3. After a successful kernel build, put these changes back.

```
# cd /usr/src/sys
# cp lib/libkern/libkern.h.orig lib/libkern/libkern.h
# cp arch/sparc/conf/GENERIC.orig arch/sparc/conf/GENERIC
```

4. The boot block will need to be replaced before you can install an *ELF* kernel. To do this you will need to boot single user mode[12]. Run QEMU as above but add *-prom-env 'auto-boot?=false'* to the command line flags, then:

```
boot disk:a -s
<system boots>

# fsck -a
# mount -a
```

See the *installboot* manual page for more details.

[12] or you can build a kernel with options INSECURE and boot that.

8 Going forward

The NetBSD project has always focused on clean code and portability. During the time frame that I have examined in this paper, the project team were thinking about the portability of the build system. The original goal of the build system project was to be able to build NetBSD on any relatively modern standards compliant Unix-like system with a standard C compiler. This has been a very successful project and probably one of the largest projects written in POSIX-compliant shell.

NetBSD 1.6 introduced the *build.sh* cross-compilation build system [4]. What *build.sh* does is setup a toolchain consisting of *make*, the compiler, assembler, linker and other tools to build NetBSD. Today's version works on multiple operating systems (Linux, Darwin, OpenBSD, FreeBSD) and it output binaries for all of the supported NetBSD architectures.

The first thing it does is builds the toolchain for the chosen target architecture. The toolchain is installed in a separate directory to the host machine's binaries. Then *build.sh* builds the libraries and user-land binaries. It can also build kernels and the full release media for the operating system.

There are many obvious advantages to this. Firstly it is possible to build all of the releases from one fast machine. Before *build.sh*, each port master had to build a release and the project had to wait for the slowest architectures to complete. Secondly, it is possible to target embedded environments easily without having a development system onboard. Thirdly, adding new architectures is made easier for the same reason. A final reason to mention is that a daily build of the development version can be done quickly for all architectures to ensure that there are no build errors.

In this paper we saw how to upgrade an earlier version by building a toolchain and then use it to build the binaries. In the future, if the NetBSD project decided to change binary format again, the process would be much easier because *build.sh* would handle it without being intrusive to the host system.

If you have understood the steps in this paper, you will be able to understand the ideas in *build.sh*. In newer versions of NetBSD, it is possible to upgrade the system using *config* and *make* with the sources, but it is far more convenient to use *build.sh* these days.

Here is the power of *build.sh*. Download the latest sources for NetBSD and in the source directory you can build an entire release for your current system or build a kernel. There is no need to change any tools on the host system - *build.sh* does the work for you, provided that your system is recent enough.

```
# ./build.sh release
# ./build.sh kernel=GENERIC
```

If you do not have root access on the system and you can build without privileges to a directory that you have access to. For example:

```
% ./build.sh -O ~/objects -U release
```

And you can choose any architecture that NetBSD supports as a target. So to build a complete release for *alpha*:

```
% ./build.sh -O ~/objects -U -m alpha release
```

Efforts have been made to move towards a compiler with a license nearer to the BSD license, but the broad platform support of *gcc* has kept it as the compiler of choice in the NetBSD tree.

At the time of writing, all of the supported NetBSD platforms are *ELF* based, with the possible exception of some requiring a differently formatted kernel file in order to boot. There is one platform, *sun2*, that does not support dynamically linked libraries but all the others do.

From the latest *share/mk/bsd.own.mk*:

```
# OBJECT_FMT:           currently either "ELF" or "a.out".
#
# All platforms are ELF.
#
OBJECT_FMT=     ELF
```

Unfortunately some platforms have been discontinued due to the lack of compiler support. For example the *pc532* platform [22] (of which there are only approximately 100 machines manufactured [13]) uses the ns32k chipset which is not supported by an up to date version of *gcc*. Sadly this platform was never converted to *ELF* before it was discontinued [14].

We hope that this paper has contributed to your understanding of binary formats but has also given you an insight into a working operating system from over 20 years ago.

9 Acknowledgements

The author would like to thank Simon Burge, Perry Metzger, Jason Thorpe and Christos Zoulas for their help with the history on this paper. The upgrade method described in the paper was developed by Christos in an early version of the NetBSD *ELF* How-to [19]. Also thanks to Craig Buckler who made some useful suggestions and Arrigo Triulzi who painstakingly ran all the commands to check nothing was missed.

10 References

[1] a.out binary format, Wikipedia

[13] One of the NetBSD development team still regularly boots a pc532 machine, although he occasionally has to fight the SCSI card to get it to work. It runs NetBSD 3.99.16.

[14] see r1.489.2.4 of *share/mk/bsd.own.mk*.

[2] a.out format manual page, The NetBSD Foundation

[3] Automated Test Framework, Wikipedia

[4] build.sh: Cross-building NetBSD, Luke Mewburn & Matthew Green

[5] BSD Licenses, Wikipedia

[6] Comparison of executable file formats, Wikipedia

[7] *Ed Mastery: The Standard Unix Text Editor*, Michael W. Lucas, Tilted Windmill Press, 2018.

[8] ELF format manual page, The NetBSD Foundation

[9] Export of cryptography from the United States, Wikipedia

[10] GCC manual, GNU Operating System Project.

[11] GNU General Public License v2, GNU Operating System Project

[12] Intel 486DX2 PC boot messages, Chris Pinnock

[13] *Linkers and Loaders*, John R. Levine, Morgan Kaufmann Publishers, 2000

[14] *Make linux ELF binaries work (11/6/95)* src/doc/CHANGES.prev, The NetBSD Project

[15] NetBSD 1.2 release announcement, The NetBSD Foundation

[16] NetBSD 1.3 release announcement, The NetBSD Foundation

[17] NetBSD 1.4 release announcement, The NetBSD Foundation

[18] NetBSD 1.5 release announcement, The NetBSD Foundation

[19] NetBSD ELF FAQ (1999 version archived), The NetBSD Foundation

[20] NetBSD History, The NetBSD Foundation

[21] NetBSD packages collection, The NetBSD Foundation

[22] PC532 platform, Wikipedia.

[23] QEMU: A generic and open source machine emulator and virtualizer,

[24] *Solaris Internals: 2nd Edition*, McDougall and Mauro, Sun Microsystems/Pearson, 2007

[25] *The Design and Implementation of the 4.4BSD Operating System*, McKusick, Bostic, Karels and Quarterman, Addison-Wesley, 1996

[26] The Development of the C Language, Dennis M. Ritchie, 1993.