Université de Montréal

A Fully Reproducible C Toolchain Rooted on POSIX Shell

par

Laurent Huberdeau

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

31 Août 2025

A Fully Reproducible C Toolchain Rooted on POSIX Shell

présenté par

Laurent Huberdeau

Résumé

Les attaques de chaîne d'approvisionnement sont de plus en plus communes et sont notamment difficiles à détecter. La compilation reproductible (reproducible builds) est une manière de prévenir ces attaques, en permettant à quiconque de recréer les artefacts de compilation de manière exacte (bit à bit) à partir du code source. Or, les outils de compilation nécessaires pour recréer ces artefacts sont à leur tour vulnérables à ces attaques. Ainsi, une solution complète au problème de chaîne d'approvisionnement doit inclure la compilation reproductible de tous ces outils.

Avec ce problème comme motivation, nous démontrons l'utilité des shells POSIX, disponibles sur la grande majorité des ordinateurs, comme point de départ pour la compilation reproductible. Pour ce faire, nous présentons `pnut-sh`, un compilateur C vers POSIX shell écrit en C dont la sortie est conçue pour être lisible. Puisque le compilateur peut s'auto-compiler, il est donc possible de produire un script implémentant un compilateur C qui ne nécessite qu'un shell conforme à la spécification POSIX tels que `bash`, `ksh`, `zsh`, etc. Ensemble, `pnut-sh` et le shell servent alors de point de départ pour la compilation d'une série d'outils de plus en plus complets jusqu'à atteindre une version récente du GNU Compiler Collection (GCC). GCC peut alors servir de fondation pour tout autre outil.

Afin de pouvoir passer de l'environnement limité du shell au reste du système d'exploitation, nous présentons également `pnut-exe`, un compilateur C plus complet produisant des exécutables x86, ainsi qu'une implémentation sur mesure de la bibliothèque standard C depuis lesquels le Tiny C Compiler (TCC) peut être compilé. TCC peut ensuite être utilisé pour amorcer GCC. Le résultat est une chaîne de compilation produite exclusivement à partir d'un shell et de fichiers source lisibles.

Nous discutons de comment compiler reproductiblement TCC à partir du shell POSIX, le niveau de support pour le langage C requis pour atteindre cet objectif, l'architecture des deux versions de pnut, la génération de code shell POSIX portable et suffisamment performant, et finalement, l'empaquetage et la distribution des fichiers servant au processus de compilation reproductible de TCC.

Mots clés: Compilation reproductible, auto-amorçage, C, POSIX shell

Abstract

Software supply chain attacks are increasingly frequent and can be hard to guard against. Reproducible builds ensure that generated artifacts (executable programs) can be reliably created from their source code. However, the tools used by the build process are also vulnerable to supply chain attacks, so a complete solution must also include reproducible builds for the various compilers used.

With this problem as our main motivation, we demonstrate that the widely available POSIX shell can serve as the only trusted pre-built binary for reproducible builds by presenting `pnut-sh`, a C to POSIX shell transpiler written in C that generates human-readable shell code. Because the compiler is self-applicable, it is possible to distribute a human-readable shell script implementing a C compiler that depends only on the existence of a POSIX-compliant shell such as `bash`, `ksh`, `zsh`, etc. Together, `pnut-sh` and the shell serve as the seed for a chain of builds that create increasingly capable compilers up to the most recent version of the GNU Compiler Collection (GCC), which is a convenient basis to build any other required tool in the toolchain.

To bridge from POSIX shell to the rest of the operating system, we also present `pnut-exe`, a more fully-featured C compiler that targets x86 and its bespoke C standard library implementation from which the Tiny C Compiler (TCC) can be compiled. In turn, TCC can then be used to bootstrap GCC. The end result is a complete build toolchain built only from a shell and human-readable source files.

We discuss how TCC can be reproducibly built from POSIX shell, the level of C language support needed to achieve this goal, the architecture of both versions of pnut, the generation of portable and performant POSIX shell code from C, and finally, the packaging and distribution of the files used in the reproducible build process.

Keywords: Reproducible builds, bootstrapping, C, POSIX shell

Contents

List of tables

List of figures

Remerciements

Je tiens à exprimer ma gratitude à tous ceux qui ont contribué à ce projet, de près ou de loin, par leur soutien, leurs conseils ou leur expertise.

Je remercie tout particulièrement mon directeur de recherche, Marc Feeley, pour son soutien tout au long du projet, et plus généralement pour sa disponibilité et grande générosité tout au long de mon parcours universitaire.

Je remercie également mes collègues du laboratoire de recherche sur les langages de programmation, Marc-André, Olivier, Léonard et Cassandre. Travailler à vos côtés a été un réel plaisir.

Je remercie mes amis, Alex et Xavier, pour leur enthousiasme et leur soutien tout au long de ce projet, et ma famille pour leur soutien indéfectible, en particulier Jacqueline et Sylvain, qui ont été et sont toujours là pour m'encourager.

Je ne pourrais terminer autrement qu'en remerciant ma conjointe, Irene, qui a su m'accompagner tout au long de ce parcours, me soutenant dans les moments difficiles et célébrant les réussites à mes côtés. Sa présence a été une source de motivation et d'inspiration constante.

Chapter 1

Introduction

Reproducible builds contribute to the integrity of the software supply chain by providing a way to reliably recreate software artifacts, typically executable programs, from their source code. To do so, reproducible builds require starting from a set of pre-built binaries, including compilers and other build tools. However, pre-built binaries are susceptible to *trusting trust* attacks [42], in which a compiler is compromised to inject malicious code into the programs it compiles and to perpetuate this covert behavior in other compilers it builds. As a result, code reviews alone are insufficient to ensure the integrity of the compiled programs; the pre-built binaries used must also be trustworthy. In this thesis, we explore the use of the POSIX shell [38] as the only trusted pre-built binary for reproducible builds. This peculiar choice is motivated by the exceptional portability and widespread availability of POSIX shell implementations, making it the ideal trusted binary for this role:

- It was standardized over 30 years ago and has multiple conforming implementations, including Korn Shell (`ksh` [5]), Bourne-Again Shell (`bash` [17]), Z Shell (`zsh` [16]), Almquist Shell (`ash` [2]), Debian Almquist Shell (`dash` [26]), Yet Another Shell (`yash` [25]) and Oil Shell (`osh` [6]).
- It is cross-platform, with implementations on all the major operating systems (Linux, macOS, and Windows), including old versions of those operating systems.
- It is readily accessible, as many operating systems have a POSIX shell preinstalled. On Windows, the Windows Subsystem for Linux (WSL) or one of the many `bash` packages for Windows can be used.
- It is a widely used standard for scripts controlling critical aspects of the operating system, so it is unlikely to evolve in a backward incompatible way.

The execution speed and portability of the generated shell scripts are clearly important issues. One of our contributions is to find ways to implement certain essential C language features, such as pointers and file I/O, that are reasonably efficient across the various shell

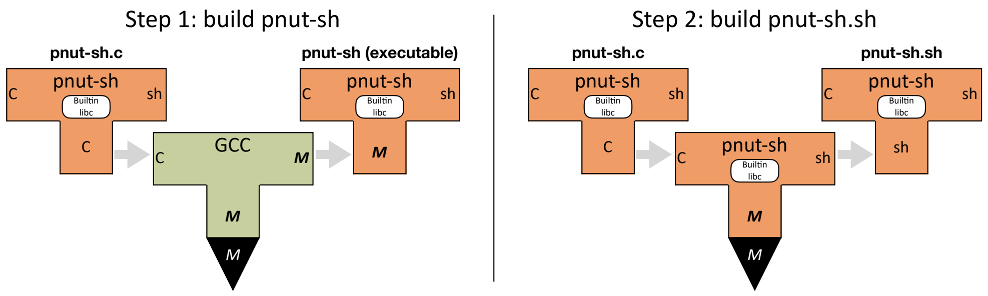


Fig. 1.1. The file `pnut-sh.sh` is created by a two-step bootstrap procedure. The file `pnut-sh.c` is first compiled by an existing C compiler, such as `gcc`. The executable compiler `pnut-sh` that is produced can then be used to compile `pnut-sh.c` again, creating `pnut-sh.sh`. Orange is used for the `pnut-sh` compiler, which compiles C to POSIX shell. The inverted black triangle represents a machine of type M, whose architecture and OS don't matter here.

implementations. Moreover, the readability of the generated scripts is an issue our work addresses. We have developed `pnut-sh`, a C *transpiler* that preserves the structure of the C source code and produces human-readable and portable POSIX shell scripts from C source code.

Another one of our contributions is demonstrating the viability of using POSIX shell as the starting point of reproducible builds. To do so, we have developed `pnut-exe`, a C99 compiler that produces binary executables and that can be hosted by `pnut-sh`, connecting the shell to the rest of the operating system. `Pnut-exe` is able to compile the Tiny C Compiler (TCC) and our bespoke C standard library implementation, making it possible to bootstrap a complete build toolchain built only from a shell and human-readable source files.

Both versions of pnut are written in C and can be hosted by `pnut-sh`. This self-compilation produces the `pnut-sh.sh` shell script, which is a portable version of the `pnut-sh` compiler. The two-step procedure to create `pnut-sh.sh` using an existing C compiler is shown in Figure 1.1. Once created, `pnut-sh.sh` can serve as the seed of a reproducible build process that goes on to compile `pnut-exe` and TCC, on any system with a POSIX shell, and with no requirement for a preinstalled C compiler or other tool. This is illustrated in Figure 1.2 and Figure 1.3 using the traditional “T” notation [1] for compilers to indicate the source/target/host language triplet.

Pnut's shell backend was designed to produce shell code that is easy for humans to understand and review. Consequently, a motivated software developer can inspect the scripts generated by `pnut-sh`, including `pnut-sh.sh` itself, to verify that it does not contain malicious code. The scripts generated by `pnut-sh` avoid shell constructs that would hinder this auditing process. This makes `pnut-sh.sh` particularly interesting as a trusted seed for *reproducible builds*.

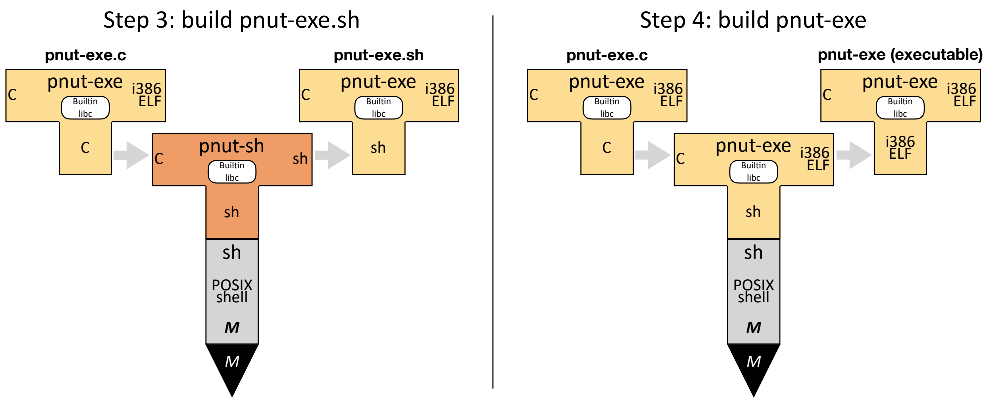


Fig. 1.2. `pnut-exe` can be bootstrapped by first compiling it to shell (`pnut-exe.sh`) using `pnut-sh.sh`, then recompiling `pnut-exe` with itself to obtain an executable (`pnut-exe`). Yellow is used for the `pnut-exe` compiler, which compiles C to native code.

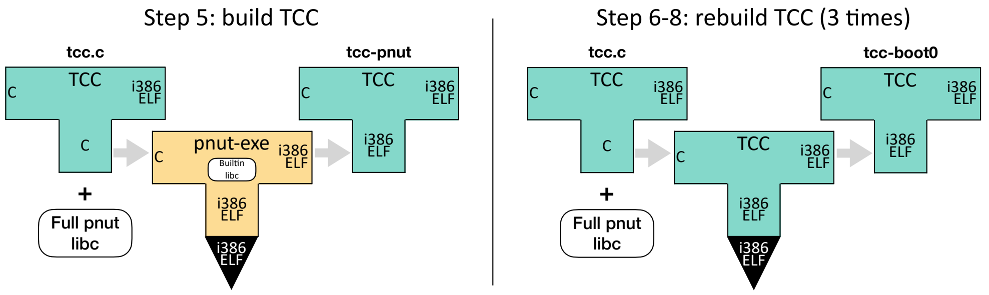


Fig. 1.3. `tcc` can be bootstrapped by first compiling it with `pnut-exe` (`tcc-pnut`), then with itself until a fixed point is reached (`tcc-boot0`, `tcc-boot1` and finally `tcc-boot2`). Note the use of the full pnut libc in addition to the built-in libc. Teal is used for the `tcc` compiler, which compiles C to native code. In this figure, the inverted black triangle represents an x86 Linux machine, the architecture and OS currently supported by `pnut-exe`.

In this thesis, we extend the design of pnut, initially described in our original paper [34], to make it a practical tool for reproducible builds. The lessons learned about portable and efficient shell scripting in the context of compiling C are likely to apply to compilers for other languages that generate portable shell scripts.

The thesis is organized as follows:

- Chapter 2 begins by introducing background material that is useful in understanding this thesis. This includes an overview of reproducible builds, the trusting trust attack, diverse double-compilation, and existing solutions to these problems. It also introduces the POSIX shell language and its history to bring the reader up to speed on the relevant aspects of the language.

- Chapter 3 makes the case for using the POSIX shell as a trusted pre-built binary for reproducible builds, justifying it through the use of diverse double-compilation. It presents `pnut-sh` and `pnut-exe`, and how they can be used to bootstrap a fully working version of the Tiny C Compiler (TCC). In addition, a solution based on POSIX shell for the packaging and distribution of the files required for the reproducible build process is provided.
- Chapter 4 describes and motivates the design choices we made in the architecture of `pnut-sh`, `pnut-exe` and of the C bespoke standard library.
- Chapter 5 goes in depth over the transpilation of C to POSIX shell code, and how to generate portable and performant shell code.
- Chapter 6 evaluates the performance of pnut and of the generated shell code on various shells, and compares our approach to other reproducible builds of C toolchains.
- Finally, Chapter 7 discusses potential improvements and extensions to the work presented in this thesis, with a focus on improving the practicality of pnut for reproducible builds.

We provide pnut, its source code and scripts to reproduce our results in the pnut GitHub repository [18] and website [33], and the project's README is included in Appendix A for convenience.

Chapter 2

Background

This chapter goes over the relevant background for understanding the contributions of this thesis. Techniques used for bootstrapping are presented, with a focus on reproducible builds and diverse double-compilation, applied to the C ecosystem. Finally, an overview of the POSIX shell language is provided. Basic knowledge of the C programming language [28] is assumed, as the POSIX shell language is presented in contrast to C.

2.1. Reproducible Builds and Bootstrapping

A *reproducible build* is a process of building software in a way that ensures that the resulting executable programs can be reproduced bit-for-bit. To achieve this, the source files must be the same for every build and the compiler used must be deterministic. Moreover, the compiler version must be fixed, as different versions can generate different outputs.

Ideally, the compiler used would itself be built reproducibly, which brings the question of how to build the compiler. Historically, this was seen just as a technical hurdle which was to be overcome via something called a *bootstrap* but whose actual details were largely considered as irrelevant, as long as it ended up delivering a functional compiler.

2.1.1. Bootstrapping Compilers

In an abstract sense, bootstrapping is the process of constructing a system from a smaller or simpler system. To bootstrap a compiler, a minimal version of the compiler is created first, which can then be used to build more complex versions of the compiler. This minimal version is called the *bootstrap compiler* and is written in a language for which a compiler already exists and serves as the starting point for the bootstrap process. The bootstrap process can be described in stages, with each stage building on the previous one, until the final version of the compiler is produced:

- Stage 0 prepares an environment from which the bootstrap compiler will execute. The tools available in the environment determines which languages the bootstrap compiler may be written in. The C language is commonly used for bootstrap compilers, as C compilers are included by default on most systems and don't impose too many constraints. Environments without compilers can also be used. In that case, the bootstrap compiler must be written in machine code compatible with the architecture of the processor.
- Stage 1 produces the bootstrap compiler from the environment, giving some level of support for the language that is being bootstrapped. Thus, the compilers of subsequent stages can be written in the same language they support.
- Stage 2 uses the bootstrap compiler to build a first version of the compiler. This compiler is functionally equivalent to the final compiler, but its code may not be as efficient as the final compiler's code.
- Stage 3 is the final compiler, which is built from the semantically equivalent but bit-wise different stage 2 compiler. The final binary supports the full language and its code is as efficient as the final compiler can generate.

  As a last step, the stage 3 compiler may rebuild itself to perform the *bootstrap test* as a sanity check that the bootstrap process and compiler are correct. The binary obtained from this later compilation should be bit-wise identical to the stage 3 compiler. While this test is not a guarantee that the compiler is correct, failing it is a strong indication that the bootstrap process or the compiler is broken.

Since the bootstrap compiler is discarded after the bootstrap, it is often much simpler than the final compiler, with fewer features and optimizations. A common approach is to use a simple interpreter as the bootstrap compiler since it is much cheaper to implement than a full compiler. Bootstraps don't always follow the above stages however and are often closer to a dark art than a science, with various ad-hoc workarounds, as documented in [35, Appendix C].

As an example, when the bootstrap compiler does not implement the full language, either because the language is too extensive, or because the language has evolved since the bootstrap compiler was written and it wasn't kept up-to-date, extra stages must be added with older versions of the compiler used to span between the different versions of the language.

This can become a problem when the compiler makes use of new language features as they are added, especially if done without an intermediate version where the feature is supported but not used in the compiler implementation. When this happens, the cycle must be resolved either through patching the version at fault to remove the use of the feature, or by extending a prior version to support the feature without using it in its implementation.

With time, more and more stages must be added to the bootstrap, increasing the time to bootstrap, as well as requiring the distribution of many versions of the compiler as source code as part of the bootstrap process. A common solution to this problem is to enshrine a specific binary of the compiler as the bootstrap compiler (effectively making the stage 0 language machine code), and have it as an input to the build process, shortcutting the earlier stages. This approach is generally less ad-hoc but can still be delicate, as discussed by Appel [3].

This solution is not without its drawbacks, as distributing and keeping track of binaries is more difficult than distributing source code. It also fundamentally undermines the ideals of Free Software [40] since it requires distributing precompiled code (basically a "binary blob") alongside the source code. Support for new platforms can also be a challenge, as the binary bootstrap compiler and whatever tools used to recreate it must be ported to the new platform, that is, assuming the knowledge of how to create the binary wasn't lost to time. Cross-compilation may be used to overcome this problem, but it itself comes with its own set of difficulties and requires multiple machines to be used for the bootstrap. These issues point to the importance of being able to bootstrap a compiler from source and having a clear and well-defined bootstrap process.

2.1.2. Reproducible Builds

A step in the right direction is to use a *reproducible build* process, which aims to ensure that the same source code will always produce the same binary. This can help mitigate some of the risks associated with bootstrapping compilers, as it allows for greater transparency and verifiability of the build process. For a build to be reproducible, a deterministic compiler, combined with a fixed set of inputs, are all that is required. In practice however, achieving reproducibility requires a lot care, as the set of inputs extends past the source code and build tools. Inputs can include file system metadata, file paths of source files and system libraries, timestamps (from the use of macros and system calls) and environment variables. Compilation options and the order of the compilation steps can also affect reproducibility. Finally, non-determinism can manifest itself in subtle ways, for example, from the use of data structures without a fixed ordering of elements, such as hash tables using the address of elements as keys when address space layout randomization (ASLR) is used.

Reproducible build is not a new idea, with GNU tools being made reproducible as early as the early 1990s, as described by John Gilmore [31]. The idea was brought back a few times in the Open Source community, but without much success as the tools used to build software were not designed with reproducibility in mind and the advantages of bit-for-bit reproducibility were not fully appreciated. Interest picked back up with Bitcoin, whose users needed a way to ensure they were running a safe version of the software. This led to the development of Gitian, a tool to build and distribute reproducibly verified binaries [11]. The interest in reproducible builds has since grown, with the Debian project reaching in 2022 [36] 100% reproducibility for its `essential` and `required` package sets on amd64 and arm64, and in 2025 [7], 100% reproducible system images for its `bookworm` release. Other distributions have followed suit, and are pursuing similar goals, such as Fedora [19] and NixOS, with the latter being particularly focused on reproducibility. The *Reproducible Builds* project [30] is a community effort to improve the state of reproducible builds in ecosystem.

Reproducible package managers are also becoming more common, with Guix [9] and Nix [12] being the most notable examples. In these systems, the build process of packages is described in a declarative and composable way, with every input to the build process being specified. Packages can also depend on other packages, which are in turn described in a similar way, allowing for a fully reproducible build process.

Utilities to help with reproducible builds have also been developed, such as `diffoscope` [15], which compares two files or directories and reports the differences between them, or `strip-nondeterminism` [13], which removes non-deterministic elements from files to make them reproducible.

2.1.3. Trusting Trust Attacks

A perhaps less obvious but nonetheless important problem with bootstrapping compilers is the risk of *trusting trust* attacks. By their nature, binary blobs are more-or-less impossible to inspect, and users have to take the compiler's authors at their word that the binary they are distributing is what it claims to be. Recompiling the compiler from its source code is often seen as a way to verify that the binary is correct and matches its source code. However, this assumes that the second set of tools used for this recompilation is trustworthy, as they could themselves compromise the binary.

Ken Thompson coined the term *trusting trust* in his Turing Award acceptance speech in 1984 [42] and describes how this might be done. In his version of the attack, Ken Thompson hides two malicious pieces of code in the system compiler: one that recognizes the login utility and injects code to grant himself a secret access, and one that recognizes the compiler itself and replicates the malicious behaviors in the new compiler binary. This last piece of code is particularly insidious, as it teaches the compiler to propagate the malicious behavior to any new compiler it compiles, even when the source is clean. The solution is, as Ken Thompson puts it:

«*You can't trust code that you did not totally create yourself.* »

To totally create the code, being able to reproducibly build it is not enough, as the tools used to build the code must also be trustworthy. Taken to its final conclusion, this means that the only way to be sure that the code is clean is to bootstrap from scratch, without the use of any pre-built binary. This includes binaries that are often taken as granted, such as the shell, command-line tools like `sed` and `awk`, and even the operating system and hardware.

2.1.4. Full-Source Bootstrap

Full-source bootstraps take the idea of reproducible builds one step further by rooting the build process in a single trusted binary as a defense against trusting trust attacks. From this binary is built the rest of the system *from source*, bootstrapping a shell, system utilities and every other dependency along the way.

As far as we know, only two projects offer a full-source bootstrap: `Guix Full-Source Bootstrap` [10] and `live-bootstrap` [24]. Both projects rely on `stage0` [23], which provides `hex0`, a 190-byte long (on x86) self-hosting assembler and a collections of intermediary tools to build a simple C compiler on many architectures, all from `hex0` and source code. Because both projects follow the exact same initial steps, we focus on `stage0` in the rest of this section.

2.1.4.1. Full-Source Bootstrap: Starting from Zero. The trusted binary used for `stage0`'s bootstrap is a 190-byte binary named `hex0`. `Hex0` is a simple self-hosting assembler, reading hexadecimal nibbles and outputting the corresponding byte to a file. Lines starting with `;` or `#` are comments and are ignored, making it possible to annotate its source code with comments that are essential for the auditing process.

Because of its size and annotated source code, `hex0` can realistically be looked at and audited by a human, even if in binary form. By self-hosting, it can regenerate itself from its source code (available as Appendix B), making it possible to verify that the binary is indeed what it claims to be (whether this is done by hand or with a prebuilt `hex0` is left to the user). Figure 2.1 shows a sample of `hex0`'s assembly source code, to demonstrate its simplicity and readability relative to the level of abstraction it operates at.

For bootstraps without an operating system, the `builder-hex0` project [20] provides a minimal environment to run `hex0`, removing the need for a trusted operating system and moving the trust to the hardware.

From `hex0`, a series of more and more powerful assemblers and compilers are built: `hex1` and `hex2`, more capable versions of `hex0` with support for relative and absolute offsets, are built from `hex0` and `hex1`. The `M0` macro assembler is then built from `hex2`, and finally a basic C compiler, `cc_x86`, can then be built from `M0`. This approach was first described by Edmund Grimley-Evans for the bootstrap of his `bcompiler` project [27], and essentially

```
; Our main function
# :_start ; (0x8048054)
  58          # POP_EAX                          ; Get the number of arguments
  5B          # POP_EBX                          ; Get the program name
  5B          # POP_EBX                          ; Get the actual input name
  B9 00000000 # LOADI32_ECX %0                   ; prepare read_only
  BA 00000000 # LOADI32_EDX %0                   ; Extra sure
  B8 05000000 # LOADI32_EAX %5                   ; the syscall number for open()
  CD80        # INT_80                           ; Now open that damn file
  A3 55810408 # STORE32_Absolute32_eax &fin      ; Preserve the file pointer we were given

  5B          # POP_EBX                          ; Get the actual output name
  B9 41020000 # LOADI32_ECX %577                 ; Prepare file as O_WRONLY|O_CREAT|O_TRUNC
  BA C0010000 # LOADI32_EDX %448                 ; Prepare file as RWX for owner only (0700)
  B8 05000000 # LOADI32_EAX %5                   ; the syscall number for open()
  CD80        # INT_80                           ; Now open that damn file
  A3 59810408 # STORE32_Absolute32_eax &fout     ; Preserve the file pointer we were given
```

Fig. 2.1. Sample of `hex0` assembly source code. Full source code is available in Appendix B.

reinvents the early history of compilers, starting from a simple assembler and building up to a C compiler. This approach was later adopted for the `stage0` project.

2.1.4.2. Full-Source Bootstrap: Reaching C. `cc_x86` is the first C compiler in the `stage0` full-source bootstrap, and it is used to build the M2-Planet C compiler, a more complete C compiler written in C. From M2-Planet is then compiled Mes [21] and MesCC which are a Scheme interpreter and C compiler made specifically for bootstrapping Tiny C Compiler (TCC [14]). The Mes interpreter is written in around 5000 lines of code from a small subset of C99, making it compatible with minimal C compilers such as M2-Planet. MesCC is a C compiler written in Scheme that can be run with Mes. MesCC's C support is sufficient to compile a fork of TCC with floating-point numbers and 64-bit integers removed. From this initial version of TCC, a complete version can then be built which can then serve as the bootstrap compiler for a more traditional C bootstrap.

2.1.5. Bootstrapping a Modern C Toolchain

The C programming language underpins many critical components of modern infrastructure, with most programming languages relying on it, directly or indirectly, for their bootstrap. Given this central role, full from-source reproducible builds for C are essential to support the reproducible builds of the rest of the software ecosystem.

The open-source C ecosystem centers around two C compilers, the venerable GNU Compiler Collection (GCC) and Clang. Modern GCC is written in a mix of C and C++, this means a C++ toolchain must be first bootstrapped. Fortunately, GCC 4.7 offers a solution to this problem by being one of the last version written exclusively in C while supporting C++. This makes it essential for bootstrapping GCC from source and so version 4.7 is still maintained by the community [8]. However, because of its age, GCC 4.7 doesn't have support for newer ISA, such as RISC-V. The use of an old version of GCC poses a challenge for reproducible builds on new platforms as support for these architecture must be backported from newer GCC versions, unless using cross-compilation to overcome this hurdle which comes with its own set of issues.

Bootstrapping GCC 4.7 is no small feat, and requires a more or less complete C99 implementation – preprocessor, inline assembly, variable-length arrays, floating-point numbers and many other non trivial features. In addition to C language support, building GCC requires bootstrapping its dependencies and all of their dependencies (transitively). In `live-bootstrap`, it takes over 80 steps before GCC is compiled for the first time. To reduce the amount of work necessary to reach GCC, the Tiny C Compiler (TCC) was extended to support the full C99 standard and to compile GCC 4.7. TCC's footprint being much smaller, with around 30000 lines of code (for x86) and requiring no external dependency or build tools as it includes its own preprocessor, assembler and linker, this reduces the level of C support required to reach GCC. For reference, GCC 4.7 source code can be counted in millions of lines of code [1]. We note that Clang and LLVM cannot be compiled by TCC, and so can only be bootstrapped from GCC, as they are written in C++. This is unfortunate as every reproducible build paths go through GCC, making it a prime target for supply chain attacks. The same can be said of TCC, as there are no other small C compiler from which the C ecosystem can be fully bootstrapped from source as far as we know.

There exists other small open-source C compilers that are sufficiently complete or close to be able to compile GCC 4.7 but for which a full-source bootstrap doesn't yet exist:

- The Chibicc C [44] compiler is a small but relatively complete C11 to x86 compiler meant as a guide on how to implement a C compiler. It is accompanied by a book (currently being written) and a series of commits that incrementally builds the compiler, making it particularly easy to audit. The compiler can compile many real world programs, such as Git, SQLite and libpng, but is itself written in C11, increasing the level of C language support required to bootstrap GCC instead of reducing it.
- The cproc [29] compiler is another C11 compiler that can compile GCC 4.7 directly, making it a contender for bootstrapping GCC. Also, unlike Chibicc, it is written in standard C99 and supports amd64, arm64 and riscv64. Unfortunately, the cproc compiler depends on the QBE compiler backend, written in C99, for its code generation,

[1]Getting an accurate count is difficult, since many files are only used for certain targets or to support languages that are not needed for bootstrapping.

and an assembler. While not a deal breaker, QBE and an assembler significantly increases the footprint of cproc, increasing the amount of work required to bootstrap it from source. Still, a full-source bootstrap of GCC using cproc hasn't been explored as far as we know and could become a useful alternative to TCC.

The challenge of building GCC from source can be reduced to building TCC. While written in C99, the relatively small size of TCC means it only uses a subset of the full C99 language, reducing the complexity required from its host compiler. For the same reason, TCC is easier to patch to work around unimplemented features.

2.1.6. Diverse Double-Compilation

A perhaps more practical alternative to retracing the history of computing and descending the abstraction ladder until reaching a minimal auditable binary seed is to use *diverse double-compilation* (DDC) as described by David A. Wheeler [45]. Diverse double-compilation offers a defense against trusting trust attacks by bootstrapping a self-hosting compiler (C) from two independent compilers (A and B) like so:

(1) Compile C's source code with both compilers A and B to produce two binaries: C_A and C_B. If all goes well, both binaries should be semantically equivalent, that is, they should behave the same when run on the same input. The content of the binaries still differ ($C_A \neq C_B$) as they come from different compilers.

(2) Recompile C with C_A, and with C_B to produce C_{C_A} and C_{C_B}. Then:
   (a) If the two binaries were semantically equivalent, C_{C_A} and C_{C_B} must be bit-for-bit identical, that is, $C_{C_A} = C_{C_B}$. This means that A and B were not compromised, or compromised in the same way.
   (b) Otherwise, one of the compilers A or B was compromised or defective, and produced semantically different binaries C_A and C_B.

The technique can also be applied to gain trust in a prebuilt compiler, A, by using a second compiler, B, to compile A's source code. The C compiler in this case is simply A. Figure 2.2 shows the diverse double-compilation process for this simpler case.

By taking two independently designed and maintained compilers, A and B, the chances of them being compromised in the same way is very low. Trust in both compilers is gained by comparing them against each other, without the need to trust one more than the other. Moreover, the technique imposes no restrictions on the compilers used, a custom compiler that is optimized for ease of development and verification is sufficient, as long as it can compile the source code of the second and produce a sufficiently complete binary to finish the bootstrapping process.

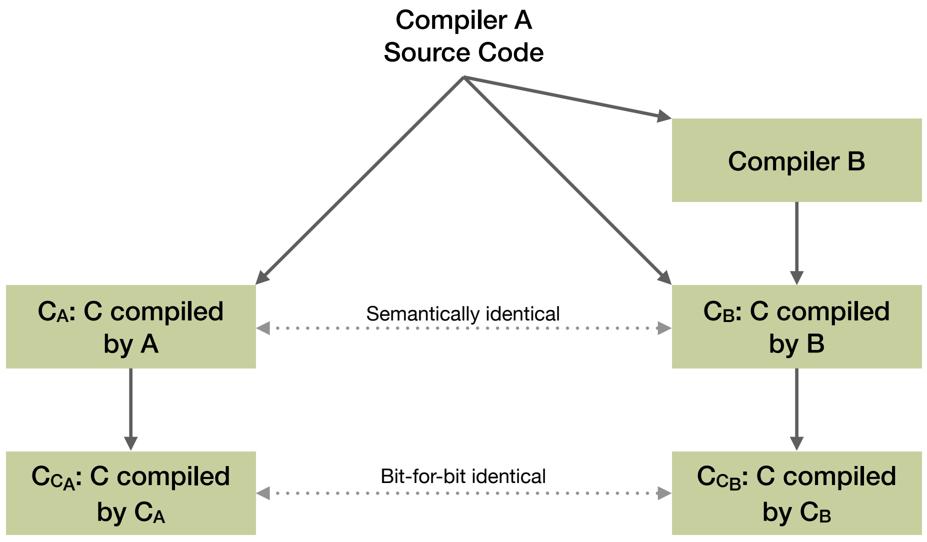


Fig. 2.2. Diverse double-compilation (DDC) bootstrapping process with prebuilt *A* compiler.

2.1.7. Use of Shells in Build Processes

How the stage 0 environment comes to be is an often overlooked aspect of reproducible builds. The security of anything is only as strong as its weakest link, and the preparation of the bootstrap environment deserves as much attention as the rest of the reproducible build.

Files used for reproducible builds are often distributed as archives (often `.tar.gz` or `.tar.xz`). Prior to the build process, the integrity of these archives is checked by hash (`sha256sum`), then they are unpacked (`tar`) and the files moved into place (`cp`/`mv`). Temporary directories may also be created (`mkdir`), and certain permissions set (`chmod`). These steps are often sequenced using the shell, another implicit dependency of the bootstrap process. These pre-built binaries are taken for granted, but are all potential attack vectors that can compromise the reproducible build process.

Full-source bootstrap solutions are not free from this problem either. Guix for example, requires the `guile-bootstrap` package in addition to the `hex0` seed. This package provides the tools to drive the reproducible build process, and includes pre-built binaries such as `bash`, `guile`, `tar`, `mkdir` and `xz`.

2.2. POSIX Shell

The POSIX shell language was standardized in 1993. It is part of the POSIX set of standards meant to provide a compatible interface between operating systems, and includes application programming interfaces (APIs), common system utilities in addition to command line shells. Shells that conform to this specification have served as the default shell on most Unix and Unix-like systems, including Linux, BSD, macOS, Windows, as well as older systems such as AIX and Solaris. The most well known implementations are as follows:

Table 2.1. POSIX shell implementations and their origin, ordered chronologically.

Shell	Author/organization	Year
Korn Shell (`ksh` [5])	David Korn (Bell Labs)	1983
Almquist Shell (`ash` [2])	Kenneth Almquist	1989
Bourne-Again Shell (`bash` [17])	Brian Fox (GNU Project)	1989
Z Shell (`zsh` [16])	Paul Falstad (Princeton)	1990
Debian Almquist Shell (`dash` [26])	Herbert Xu	1997
Yet Another Shell (`yash` [25])	WATANABE Yuki	2009
Oil Shell (`osh` [6])	Andy Chu	2016

Most of these shells are relatively old, showing the longevity of the POSIX shell standard which is still relevant to this day. This longevity also demonstrates the overall stability of the standard, with few changes to the core language over the decades, meaning scripts are not only portable across implementations but also across time. Most implementations deviate from the standard in some ways, either from bugs or through extensions that redefine part of the language or utilities. As such, for maximal portability, scripts must be not only written in the standard but in a smaller subset that shells generally implement correctly.

The POSIX shell language is relatively basic, offering few of the facilities of modern programming languages. The language takes inspiration from C, with functions being delimited by curly braces ({}), statements being delimited by semicolons (or newlines), keywords such as `if`, `while`, `break` and `continue` with equivalent semantics, as well as infix notation for arithmetic and logical operators. This similitude may be explained by the fact that early shells were written in C, and means it is possible to write shell scripts that read like C code with some effort.

That being said, the language has a strong focus on string manipulation and command execution over structured data and arithmetic, with its capabilities and idioms centered around these tasks. This is reflected in the syntax of most scripts, making them usually unrecognizable from C.

The rest of the chapter goes over the POSIX language features that are required for understanding pnut-generated code. This subset is also one we've found to be relatively well supported among shell implementations that were tested.

2.2.1. Strings and Expansion

Strings can be expressed with double quotes, single quotes, and no quotes when the string's characters have no special meaning for the shell. Shell variables are assigned string values with the syntax *name=string*.

Variables are accessed using the expansion mechanism. Commands are first expanded before being executed. Expansion is central to how the shell works and is how most operations are done. For our needs, the expansion steps are:

(1) Variable expansion

(2) Arithmetic expansion

(3) Command substitution

The `$` character introduces all these expansions. For variable expansion to happen, the syntax `$`*name* is used, or the alternative `${`*name*`}` which is useful in situations where the expansion needs to be immediately followed by a character that is valid for variable names. The value of the variable is then substituted in its place. Expansion of undefined variables is legal, and produces an empty string. Expansion also can be transformed with the following forms:

- `${#`*name*`}`: Length of the variable value
- `${`*name*`#`*pattern*`}`: Remove shortest prefix pattern
- `${`*name*`##`*pattern*`}`: Remove longest prefix pattern
- `${`*name*`%`*pattern*`}`: Remove shortest suffix pattern
- `${`*name*`%%`*pattern*`}`: Remove longest suffix pattern
- `${`*name*`-`*value*`}`: `$`*name* if non-empty, otherwise *value*

In these forms, the pattern is itself the subject of variable expansion, which is useful to match a computed pattern.

Arithmetic expansion runs second and is used to evaluate integer expressions with the syntax `$((`*expression*`))`. Section 2.2.2 explains this in more details.

Command substitution comes next and handles the syntax `$(`*command*`)` and the alternative syntax `` ` ``*command*`` ` ``. This expansion is done by running the command in a subshell, and substituting it with the standard output of the command. A subshell works like a subprocess, inheriting all characteristics of its parent. This includes variables and function declarations. However, any changes in the subshell are discarded when it ends. It is the idiomatic way to capture the output of a command in a variable for later use, for example:

```
num=42 thing=" black cat"
x="$num${thing}s"           # x = "42 black cats"
echo "length of \"$x\" is ${#x}"
hex=$(printf "0x%x" $num)  # capture output
echo "$num = $hex"          # prints: 42 = 0x2a
```

2.2.2. Arithmetic Expansion

The arithmetic expansion `$((`*expression*`))` evaluates the expression using signed integer arithmetic with at least 32 bit precision[2]. Most of the C integer arithmetic operators are allowed in the expression, with the exception of the increment and decrement operators, the `sizeof` operator and prefix `&` and `*` operators. Moreover, function calls are not allowed in expressions. Operators have the same precedence as C. This close similarity is convenient for translating C to shell.

In expressions, variables are accessed without the `$` prefix, with empty or undefined variables replaced by 0. Assignment operators such as `=` and `+=` are supported, and update the variable outside the expression, declaring it if needed. For example, `$(( (x = y = 1 << 5) > 10 ))` expands to 1 (true) and assigns 32 to the variables `x` and `y`.

A very important and perhaps less known feature of arithmetic expansion is that it can be nested. In appearance, this is not any more useful than using 1 level of expansion, but because of the order of expansion, it can be used to define variables with dynamic names, as in this example:

```
a_6=42 a_7=5 i=7
: $(( a_$i += 1 ))        # Increment a_7
echo $(( a_$((a_$i)) ))  # Output a_6, i.e. 42
```

This level of indirection is essential to implement arrays in POSIX shell as they are not supported natively. This avoids having to use a more general evaluation method such as `eval` which is considered bad security practice as it can execute any shell command. However, using arithmetic expansion to implement arrays limits the array elements to integer values.

2.2.3. Variables and Functions

Shell functions can be declared and called like so:

```
foo() {
  ...
}
```

[2]We infer this from the POSIX standard specification “Only signed long integer arithmetic is required.”

```
foo 1 2 3  # Call foo with arguments 1, 2 and 3
```

The shell does have a `return` statement to immediately return from a function, but the return value is limited to an integer exit code between 0 and 255, indicating success (zero) or failure (non-zero) of the function. Instead, to return a value from a function, it is customary to write the return value in a global variable and have the caller access this variable to retrieve the value.

Shell functions do not declare their parameters. This is because functions can take any number of arguments which are received in the function using positional parameters – the first argument is in `$1`, the second in `$2` and so on. The syntax `$#` expands to the number of arguments passed to the function. It is also possible to refer to all parameters at once using the syntax `"$@"` which expands to the function parameters separated with whitespace.

The following commands interact with the parameters:

- `set` replaces the positional parameters with the arguments it receives.
- `shift` drops the first parameter of the function and shifts the positional parameters by 1 to the left. This can be used to iterate over a function's arguments:

```
concat() {
  string=
  while [ $# -ge 1 ]; do # Iterate over params
    string="$string$1"   # Concat string and $1
    shift                # Drop the first param
  done
}

concat "abc" "def" "ghi" "jkl" "mno"
printf "The alphabet begins with $string"
```

The positional parameters of a function and `$@` are locally scoped. As we will see, this is very convenient as all other shell variables are global.

2.2.4. If, While, and Case Statements

Conditional statements use the `if` *cond*`;` `then` ... `fi` syntax, where *cond* is a command that succeeds with exit code 0 if the condition is true and fails otherwise. The `then` and `fi` keywords mark the beginning and end of the conditional block. The `else` and `elif` keywords are also supported.

The condition command can be any command including the "`[...]`" command which tests for various conditions. The following condition operators are supported: equality (`-eq` and `-ne`) and comparison (`-lt`, `-le`, `-gt`, and `-ge`) between integers, equality between strings (`=` and `!=`), as well as `-z` and `-n` to test for empty or non-empty strings. For example:

```
x=42
if [ $x -lt 10 ]; then
  echo "Small"
elif [ $x = 42 ]; then
  echo "42!"
else
  echo "Nothing special"
fi
```

While loops use the syntax `while` *cond*`; do ... done`, with the condition command working like for `if`. The `do` and `done` keywords mark the beginning and end of the loop body. Similarly to C, the `break` and `continue` commands respectively exit the loop or go to the next iteration.

```
while [ $i -lt 10 ]; do
  echo "i is $i"
  : $((i += 1))
done
```

The form `case` *string* `in;` *pattern1*`)` *cmds* `;;... esac` is used for multiway branching. The commands associated with the first pattern that matches *string* are executed.

2.2.5. Input and Output

The POSIX shell's standard ways to do I/O include:

- `echo`: Outputs its arguments to the standard output with a newline at the end.
- `printf` *format arg1 arg2*...: Formats a string and outputs it to the standard output. Supports `%s`, `%d`, `%o`, and \octal_num in the *format* string like C's `printf`.
- `read` *var1 var2*...: Reads a line from the standard input, splits it on the delimiters specified by the `IFS` (Internal Field Separator) shell variable and stores the words in the variables passed as arguments. The `-r` option disables "backslash escaping".

A noteworthy limitation of the `read` command is that it can't read null bytes, so it isn't possible for the shell to consume binary input. However, `printf` can output any byte, so it can create binary output.

The output of commands can also be redirected to other file descriptors, such as standard error or a file. Files can be opened for reading and writing using the `exec` command and the `<` (read from), `>` (write to) and `>>` (append to) redirection operators, and file descriptors can be referred by their number prefixed with `&`. To close a file descriptor, the `exec` command with the `<&-` operator can be used. A text file `a.txt` can be copied to the file `b.txt` like this:

```
exec 3< a.txt # Open a.txt in read mode as fd 3
exec 4> b.txt # Open b.txt in write mode as fd 4

IFS=                        # Don't split line
while read -r line <&3; do  # Read line from a.txt
  printf "%s\n" "$line" >&4 # Write line to b.txt
done;

exec 3<&-  # Close files a.txt and b.txt
exec 4<&-
```

Chapter 3

Bootstrapping TCC from Shell

This chapter covers how a modern version of TCC can be bootstrapped from a POSIX shell. It presents pnut, a compiler we've developed to bridge the gap between the existing C reproducible build ecosystem and the POSIX shell. The complete bootstrapping process is described, covering the compilers that are used to reach TCC, as well as how the build environment itself is bootstrapped.

The version of TCC used in this chapter is Bootstrappable TinyCC [22], a bootstrap-friendly fork of TCC that is used by the Mes project [21]. This fork is based on a commit from May 2017, which is very close to the last TCC release, version 0.9.27, released in December 2017, and so the bootstrapping techniques likely apply to the upstream 0.9.27 version.

3.1. Diverse Double-Compilation and POSIX Shell

As described in the background chapter, diverse double-compilation (DDC) can be used to counter trusting trust attacks and make reproducible builds safer while still being practical and allowing the use of pre-built binaries. The only condition is that the reproducible build process can be started from two sufficiently independent pre-built compilers. The acquisition of these compilers, and what constitutes truly independent compilers, is a problem that is not solved by DDC and is left for authors of reproducible build processes to solve. We argue that the POSIX shell is likely one of the ideal platforms to use for diverse double-compilation in general because of the following attributes:

- It has multiple conforming implementations, including Korn Shell (`ksh` [5]), Bourne-Again Shell (`bash` [17]), Z Shell (`zsh` [16]), Almquist Shell (`ash` [2]), Debian Almquist Shell (`dash` [26]), Yet Another Shell (`yash` [25]) and Oil Shell (`osh` [6]).
- It is cross-platform, with implementations on all the major operating systems (Linux, macOS, and Windows), including old versions of those operating systems.

- It is readily accessible, as many operating systems have a POSIX shell preinstalled. On Windows, the Windows Subsystem for Linux (WSL) or one of the many `bash` packages for Windows can be used.
- It is a widely used standard for scripts controlling critical aspects of the operating system, so it is unlikely to evolve in a backward incompatible way.

Combined, these attributes make sourcing independent POSIX shell implementations very easy, and the large number of independent implementations means that DDC can be performed multiple times to gain very high confidence in the resulting reproducible build process.

Historical versions of the POSIX shell can also be found in operating system images, making it possible to perform DDC on older pre-built versions of shells, which are more likely to be independent from each other and uncompromised, as they come from a time before the internet and before supply chain attacks were common. Therefore, the shell's diversity does not solely arise from its many implementations, but also from its age and ubiquity.

Additionally, trusting trust attacks target specific compilers and reproducible build processes since they must recognize specific characteristics of those processes to determine when to inject malicious code and when to propagate the compiler-specific malicious code. This lack of flexibility can be exploited by establishing new reproducible build processes working in novel and unusual ways; because the binary is fixed, the trusting trust attack it may contain cannot possibly recognize all the ways a reproducible build process can be set up, making it possible to avoid triggering the trusting trust attack or triggering it in ways that does not influence the end result. This is particularly relevant to POSIX shell, as it has never been used to host a C compiler to the best of our knowledge, meaning that trusting trust attacks against pnut cannot possibly exist in binaries that predate it.

More importantly, our choice of the POSIX shell as the base for DDC is also motivated by its use in the preparation of bootstrap environments. The shell is already an integral part of most reproducible build processes, and so is implicitly trusted to some extent. Since the shell is often the interface between the system and the user, a malicious shell implementation could compromise the entire build process before it even starts.

All these factors combined make the POSIX shell a unique and particularly apt platform for diverse double-compilation.

3.2. The Pnut Compiler

Using POSIX shell as the only binary for a reproducible build, or more generally as a platform to host a compiler, is not something that has been done before. Since TCC is written in C99, a C compiler that can be hosted by the shell is needed to bootstrap TCC. To go from POSIX shell to C requires the development of a compiler adapted to the constraints of common shell implementations. For this reason, we have developed two compilers that together form a chain that extends from POSIX shell to C. These compilers are:

- **pnut-sh**: A compiler that compiles C code to human-readable POSIX shell code.
- **pnut-exe**: A compiler that compiles C code into executables (x86 Linux and macOS) that can be hosted by **pnut-sh**.

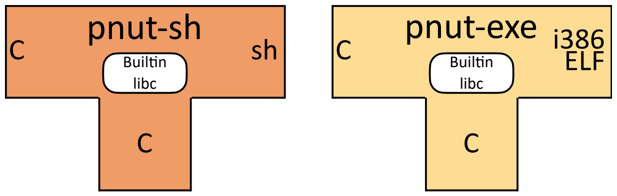


Fig. 3.1. “T” notation [1] for `pnut-sh` and `pnut-exe`.

The primary motivation for having two compilers is that implementing complex programs in POSIX shell is tedious and error-prone. Having a C to POSIX shell compiler makes it possible to do most of the development in C, a language that is known by most programmers, unlike POSIX shell. Since TCC cannot be compiled directly to shell because it requires primitives that are not available on that platform and would be too slow anyway, an intermediate C compiler (`pnut-exe`), that can compile TCC to an executable, is needed. Additionally, this choice allows `pnut-sh` to stay minimal, a crucial property for a bootstrap seed, while `pnut-exe` can be more feature-rich.

The choice of C for both compilers brings several advantages:

- Both compilers can share the same frontend, reducing the amount of code that needs to be maintained and audited.
- C is a language that is known by many programmers, making pnut's code accessible to a large number of developers and auditors.
- POSIX shell and C are syntactically similar and have a similar execution model. This makes the transpilation from C to shell relatively straightforward as described in Section 2.2.

A drawback of this approach, however, is that `pnut-sh` must be precompiled to a shell script (named `pnut-sh.sh`) through a process that makes use of an existing C compiler, which is not in the spirit of reproducible builds. However, `pnut-sh` is designed to produce human-readable shell scripts, and so `pnut-sh.sh` is intended to be audited by the user. `pnut-sh.sh` is also what is distributed along with other source files and is used to bootstrap the build process. The generation of human-readable shell code is a key feature of pnut, as it allows the use of pre-built binaries in the preparation of `pnut-sh.sh` without introducing the risk of trusting trust attacks. The precompiled script can be trusted as long as the user can verify that it does not contain malicious code, which is made easier by the human-readable nature of the generated shell code. Also, the original C source code may be included as shell comments in the generated shell script to make the code even easier to audit (See Figure 6.1).

Not all C99 features map naturally to human-readable POSIX shell, and so `pnut-sh` implements a comfortable subset of C99 that is sufficient for compiling `pnut-exe`. It is also for this reason that `pnut-exe` is needed, as some features of C99 used by TCC are not supported by `pnut-sh`. `Pnut-sh` also comes with a small built-in standard library, giving access to file I/O and memory allocations. `Pnut-exe` comes with a larger built-in standard library that implements the system calls required for the standard library functions used by bootstrap utilities and TCC. The built-in standard library is described in more detail in Section 4.2.

Both versions of pnut are written in a small subset of C. This subset is limited to:

- Function declarations, loops and conditionals.
- No pointer arithmetic, only pointer dereferencing and assignment.
- No structs or unions, only enums and arrays to represent data.
- No dynamic memory allocation using `malloc` and `free`, only statically allocated arrays.
- A few I/O primitives and standard library functions: `printf`, `fopen`, `fclose`, `fgetc`, `open`, `write` and `close`.
- Only 32-bit signed integers, characters and their pointers.
- Only one compilation unit, without header files, using a unity build instead.
- Use of simple preprocessor directives only:
  - `#define`, `#include`, `#if`, `#else`, `#endif`, `#ifdef`.
  - No self-referencing or variadic macros.

The use of a simple C subset allows pnut to be bootstrapped from much simpler C compilers and enables `pnut-sh.sh` to be short and simple, making it easier to audit. This results in a high “power-to-weight” ratio, as pnut doesn’t require much to be compiled but can compile a large subset of C99, a desirable property for bootstrapping tools. Figure 3.2 shows the C language features supported by `pnut-sh` and `pnut-exe` compared to the C language features required by TCC.

3.2.1. C Language Supported by `pnut-exe`

`Pnut-exe` extends the C99 subset above to one that is almost sufficient for TCC.

- Unsigned and fixed-width integers.
- Structures and unions, including anonymous members and flexible array members.
- Stack-allocated arrays and structures.

- Static local variables.
- The address of (`&`) operator.
- Variadic functions.
- Function pointers and indirect function calls.
- `goto` statements and `switch` with fall-through.

Well-aligned and padded structures and unions are critical as TCC uses them to read and write binary files such as ELF objects. Any discrepancy between the expected structure layout and the layout produced by `pnut-exe` would result in a different and invalid binary file.

The following features were intentionally omitted from `pnut-exe` as we deemed them too complex and not worth the effort to implement, given that TCC only uses them in a few places that can be patched out relatively easily. How to work around these missing features is described in Section 3.4.

- Floating-point numbers.
- 64-bit integers on 32-bit systems.
- Variable-length arrays (VLAs).
- Self-referential macros.
- Bitfields.
- `setjmp` and `longjmp`.

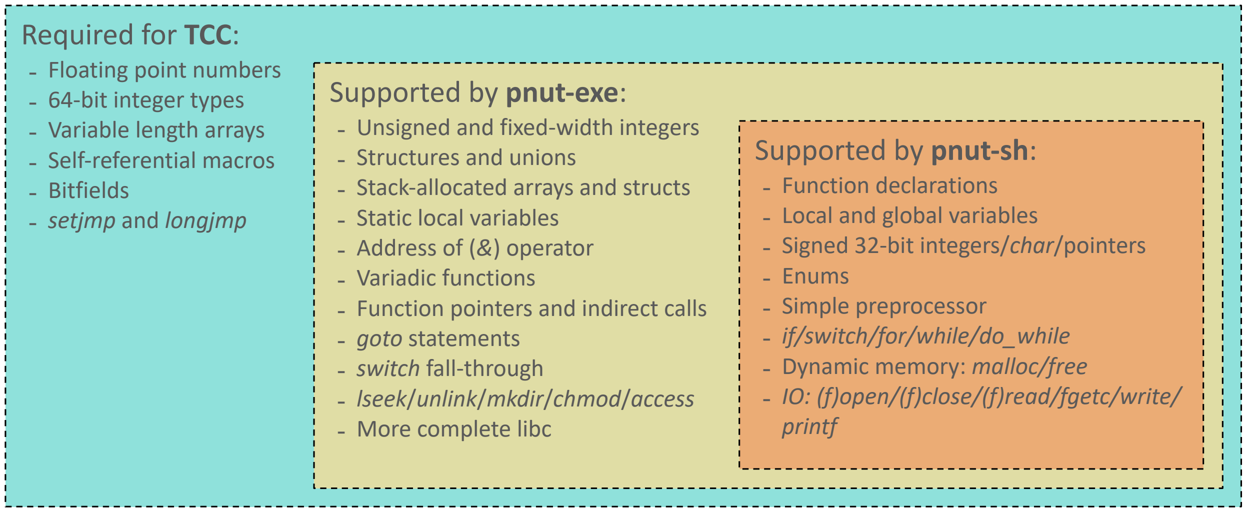

Required for **TCC**:
- Floating point numbers
- 64-bit integer types
- Variable length arrays
- Self-referential macros
- Bitfields
- *setjmp* and *longjmp*
Supported by **pnut-exe**:
- Unsigned and fixed-width integers
- Structures and unions
- Stack-allocated arrays and structs
- Static local variables
- Address of (&) operator
- Variadic functions
- Function pointers and indirect calls
- *goto* statements
- *switch* fall-through
- *lseek/unlink/mkdir/chmod/access*
- More complete libc
Supported by **pnut-sh**:
- Function declarations
- Local and global variables
- Signed 32-bit integers/*char*/pointers
- Enums
- Simple preprocessor
- *if/switch/for/while/do_while*
- Dynamic memory: *malloc/free*
- IO: *(f)open/(f)close/(f)read/fgetc/write/ printf*

Fig. 3.2. C99 language support for `pnut-sh` and `pnut-exe`, and the C99 used by TCC. From the center and moving outward, the boxes contain the features supported by `pnut-sh`, the features supported by `pnut-exe`, and the features required for TCC.

3.2.2. Alternatives to `pnut-exe`

Developing a C compiler capable of compiling TCC from scratch is a daunting task. As such, porting an existing C compiler to be compatible with `pnut-sh` was considered as an alternative to writing `pnut-exe`. However, this option was quickly discarded as the C language and the standard library supported by `pnut-sh` are relatively limited, and very few C compilers restrict themselves to this subset [1]. Thus, porting an existing C compiler to `pnut-sh` would have likely been as difficult as writing a new compiler from scratch. Moreover, the shell is a poor platform for executing large programs that allocate large amounts of memory, meaning that even if a compiler could be ported to `pnut-sh`, it would likely not be able to compile TCC in a reasonable amount of time since it wasn't designed to be run on a shell. `Pnut-exe`, whose design was heavily influenced by the constraints of the shell, takes a few minutes to recompile itself (to finally produce an executable and escape the shell's limitations), showing that compilers not designed for the shell could very well take hours to execute.

The time required for `pnut-sh` to compile these compilers is also a concern, as the compilation time of `pnut-exe.c` is already significant while containing close to the bare minimum of C features for TCC. Existing and more complete compilers could take significantly longer to compile, increasing the total bootstrap time even more.

It is worth mentioning Mes, a minimal Scheme interpreter written in minimal C and a C compiler written in Scheme that can bootstrap TCC, making it a good candidate for porting to `pnut-sh`. However, one of the goals of pnut is to produce an alternative bootstrapping path to TCC, and so relying on Mes would not be in the spirit of this work. Additionally, Mes suffers from long compilation times, which would only be made worse by porting it to `pnut-sh`, as the Scheme interpreter would now be executed in a shell environment.

3.3. Bootstrap Environment

File archives have a special place in reproducible builds, as they are used to package source code and other files together while making it easy to verify their integrity with a single checksum. The source code is then extracted, moved to the correct directories and may be patched before being compiled. After compilation, the resulting binaries are checked against checksums to ensure that the build process produced the expected binaries, and finally installed.

[1]The only C compiler we are aware of that can be compiled as-is by `pnut-sh` is `C4` [41], a very minimal C compiler that is under 500 lines of code.

All these steps require tools that are generally preinstalled on systems: `sha256sum` to compute checksums, `tar` to unpack archives, `cp` to copy files, `mkdir` to create directories and `patch` to apply patches. These tools are easily accessible from the shell, but cannot be used in our reproducible build process as they are not part of the shell binary and so are not covered by our diverse double-compilation argument. They must be bootstrapped before they can be used, so pnut comes with its own simple implementation for these tools:

- `chmod` to change file permissions.
- `cp` to copy files.
- `mkdir` to create directories.
- `sha256sum` to compute the checksums.
- `patch` to apply patches to TCC.
- `tar` to unpack `tar` archives.
- `gzip` to decompress `gzip` archives.

These tools are written in C and are compatible with pnut. They are designed to be minimal and sufficient for the bootstrapping process, favoring simplicity over completeness and performance. Once TCC is bootstrapped, it will be possible to replace these basic tools with their standard counterparts.

To simplify the distribution of the source files of these tools, they are combined into a single C file named `bintools.c`, which is compiled by `pnut-exe`. The `bintools` executable is then installed in the `/usr/bin` directory multiple times, each time with a different name to match the tools listed above. Depending on the file name, the executable will execute a different function, allowing the same binary to be used for multiple tools, similar to how the `busybox` utility works.

Bootstrapping `chmod` is particularly important, as it is used to set the execute permission on executables produced by `pnut-exe.sh` since the shell doesn't allow the creation of executable files. Bootstrapping `chmod` poses a problem, however, as it itself needs to be marked as executable to be run. To solve this, the bootstrap process assumes the existence of a writable file whose execute bit is set on the system. This file, whose content is irrelevant, is then overwritten (keeping the execute bit) with the `bintools` executable. This effectively steals the execute bit from the file and sets it on the `bintools` executable, allowing it to be run.

3.3.1. Solving the Initial Environment Problem

As explained in Section 2.1.7, the preparation of the bootstrap environment is crucial to the success of the bootstrap process, but is not part of the reproducible build process itself, and so non-trusted tools, notably the shell, are often used for this step. We refer to this as the *initial environment problem*, since the use of non-trusted tools to set up the environment introduces a risk of malicious code being executed before the bootstrap process starts, potentially compromising the entire process before it even begins. Fortunately for us, pnut explicitly depends on a POSIX shell, meaning using the shell to set up the bootstrap environment doesn't introduce any additional risks, as long as no other tools are used. Still, the shell is not sufficient to set up the environment, and merely sequences external tools that perform tasks such as creating directories, unpacking archives, changing file permissions and verifying checksums. Therefore, these tools must be bootstrapped from the shell as part of the initial environment setup.

A convenient way to solve the initial environment problem is to use `shar` archives, which are self-extracting archives distributed as shell scripts. As the format is not defined by any standard, many implementations exist, each with its own file format and options. The GNU `shar` implementation is one of the most widely used and was introduced in 1982 by James Gosling [39].

We took inspiration from the GNU `shar` implementation to create our own `shar` implementation that doesn't rely on `echo`, `test` and `sed` like GNU `shar` does. We call it **jam**, and refer to the resulting archive as a *jam archive*, often named `jammed.sh`. Figure 3.3 shows an example of a jam archive. An important consideration is that the jam archives be human-readable, so they can be easily checked for malicious code, both in the code they run and in the content of the files they create. GNU `shar` solves both by making the archive follow a particular and simple format such that a simple grep can be used to find the executable code in the archive. Data is stored in a here-document without any compression, using a single-quoted delimiter to prevent shell expansion in the here-document. Text files are stored as-is, with no encoding or escaping, and binary files are stored as uuencoded text. The bootstrap environment contains only source files (and `pnut-sh.sh`), which are all text files, so no uuencoding is needed. Still, the reproducible build community makes extensive use of archive and compressed formats such as `.tar.gz` or `.tar.xz`, and so our `jam` implementation support these formats as well to make it easier to integrate with existing reproducible build processes.

Additionally, build scripts often require the source files to be organized in directories, which would necessitate that the jam archive create these directories during the extraction process. This simple operation is a problem for us, as the archive is unpacked before the `mkdir` tool is available. To solve this, the jam archive is extracted in steps, where the first step lays the source files for `pnut-exe` and `bintools.c` in the current directory. Once `bintools` is bootstrapped, the second step can then produce the remaining files neatly organized in directories.

```
 1 #! /bin/sh
 2 
 3 pcat() {
 4   while IFS= read -r line; do
 5     printf "%s\n" "$line"
 6   done
 7 }
 8 
 9 extract_fib_c() {
10   printf "Extracting fib.c\n"
11   pcat << 'EOF3141592653' > fib.c
12 int fib(int n) {
13   if (n < 2) {
14     return n;
15   } else {
16     return fib(n - 1) + fib(n - 2);
17   }
18 }
19 EOF3141592653
20 }
21 
22 extract_fib_c
```

Fig. 3.3. Jam archive containing the `fib.c` source file, produced using the `jam.sh` script (see Appendix C). The content of the `fib.c` file appears as-is between lines 12 and 18, and is delimited by the `EOF3141592653` marker. Running this script will extract the `fib.c` file in the current directory.

An example execution of the jam shar archive is shown in Figure 3.4, where the archive is named `jammed.sh`. The archive is partially extracted during the first execution, creating the necessary files in the current directory for bootstrapping `pnut-exe` and `bintools`. The `jammed.sh` can then be re-executed to produce the remaining files in the correct directories.

3.3.2. Auditing of Jam Archives

Being human-readable is not sufficient to make jam archives safe to use. Because of their size, it may be impractical to manually audit the entire content of a jam archive before executing it. To assist with this task, we provide two simple tools. The first, named `sift.sh`, can extract specific files from jam archives, relying on the predictable format of jam archives to locate and extract their content without executing the archive script. The second, named

```
$ printf '%s\n' * # equivalent of ls
jammed.sh
$ . jammed.sh
Extracting files...
$ printf '%s\n' * # equivalent of ls
jammed.sh
bootstrap.sh  # build script to start build process
sha256sum.sh  # shell implementation of sha256sum (text files only)
pnut-sh.sh    # pnut-sh shell script
pnut-exe.c    # pnut-exe source files
fcntl.h       # /-------------------------------
setjmp.h      # |
stat.h        # |
stdio.h       # | pnut libc header files
stdlib.h      # |
string.h      # |
unistd.h      # \-------------------------------
pnut-libc.c   # pnut libc source files
bintools.c    # all-in-one chmod/cp/mkdir/sha256sum/patch/tar/gzip
```

Initial execution of `jammed.sh` that extracts files located in the current directory.

```
$ . jammed.sh
Creating directories...
Extracting files...
$ printf '%s\n' * # equivalent of ls
...         # Same files as before
TCC/        # TCC source files
libc/       # Full pnut libc (headers and source files)
```

Second execution of `jammed.sh` that extracts files located in subdirectories.

Fig. 3.4. Two-step extraction of the jam archive with the reproducible build files.

`more.sh`, is a minimal pager to view text files one page (24 lines) at a time. Each can be implemented in less than 30 lines of POSIX shell code (see Appendix D and Appendix E), allowing them to be typed by hand. With them, it is then possible to extract more complete auditing tools that can then be used to audit the rest of the archive, effectively bootstrapping the auditability of jam archives. For example, a shell implementation of the `sha256sum` tool fits in approximately 300 lines of POSIX shell code, and can be used to verify the integrity of the files in the archive, or of the full archive, against known good checksums.

3.4. Bootstrapping TCC

As explained in Section 3.2, `pnut-exe` can almost compile TCC with the following exceptions:

- Floating-point numbers.
- 64-bit integers on 32-bit systems.
- Variable-length arrays (VLAs).
- Bitfields.
- Self-referential macros.
- `setjmp` and `longjmp`.

All but the first two can be patched out of TCC, as they are only used in a few places and so can be easily replaced with equivalent code supported by `pnut-exe`. Variable-length arrays can be replaced with sufficiently large fixed-size arrays, bitfields can be replaced with regular integer types as long as the memory representation is not important which is the case for TCC, self-referential macros can be renamed to stop them from expanding indefinitely and `setjmp` and `longjmp` can be stubbed out as they are only for error handling.

However, the first two are more problematic, as TCC uses floating-point numbers and 64-bit integers throughout the codebase, making the changes required much more extensive. To find how to work around these missing features, it is important to first understand how they are used in TCC and when the code is executed.

3.4.1. 64-bit Integers in TCC

Starting with 64-bit integers, they are used in the following contexts [2]:

- In the parser, to parse and store 64-bit literals and enumeration values.
- In the inline assembly (`asm`) parser and code generator.
- In the ELF object module, to read and write ELF-formatted files.
- In the code generator and the virtual stack implementation, which performs reduction of constant expressions and converts stack-based code generation into register-based code generation.

For 64-bit literals, TCC uses very few of them, all of which can be replaced with equivalent expressions using 32-bit literals. For enumeration values, only small values that fit in 32-bit integers are used in the source code, so replacing the 64-bit types used to store the values with 32-bit types won't cause any problem.

[2]found by looking for uses of `uint64_t`, `int64_t`, `long long` and large literals in the source code

For the `asm` block parser and assembler, TCC uses 64-bit integers in the instruction opcode computation and to reduce constant expressions (typically label offset computations). Fortunately, the x86 opcode computations fit in 16-bit integers, and the inline assembly blocks used in the pnut libc are simple, with integer constants small enough to fit in 32-bit integers. This means the 64-bit types in the `asm` code generator can be safely replaced with 32-bit types.

For the ELF object module, TCC makes use of fixed-width integers in the data structures that model the ELF objects. Because the memory representation of these structures matches the ELF file format specification, it is crucial that the size and alignment of these structures be as defined by the C standard, else the ELF files produced by TCC would not be valid. Fortunately, all structures in `elf.h` that use 64-bit integers end up only being used for 64-bit ELF files, which are not relevant when targeting 32-bit systems, or in parts of the ELF file format that are not used at all by TCC, even if present in the source files, `Elf32_Move` being one such example. As a result, replacing these 64-bit integer types with 32-bit types makes no difference to TCC when targeting 32-bit systems.

Use of 64-bit integers in the code generator is more extensive and harder to patch out correctly. The code generator uses 64-bit integers for:

- Switch case values, which, like enums, can be replaced with 32-bit integers since TCC does not use large case values.
- Masks for bitfield. Bitfields are already patched out of TCC, so the code is never executed and can be removed during bootstrapping.
- Special NaN, SNaN and infinity values (`__nan__`, `__snan__` and `__inf__`) are mapped to 64-bit literals that correspond to the floating point representations of these values. These values are not used in TCC and can be removed during bootstrapping.
- In the virtual stack implementation, to store the value of constant operands and for constant expression evaluation.

The constant expression evaluation is the most delicate part, as it is used to evaluate `integer constant expressions`, used for array sizes, enumeration constants, and switch case values, making it an indispensable part of TCC's code generator, and replacing the 64-bit types with 32-bit types can lead to overflows and miscompilations. In particular, negative number literals used in 64-bit contexts are problematic, since TCC implements 64-bit arithmetic on 32-bit targets by splitting the operands into two 32-bit operands using a right shift (i.e. `>> 32`, see `lexpand` in Figure 3.6). When using `uint32_t` for `CValue.i`, right shifting produces a zero value, effectively zero-extending the negative number operand into a positive number.

```
#if HAVE_LONG_LONG
uint64_t l1 = c1 ? v1->c.i : 0; // Value of left operand
uint64_t l2 = c2 ? v2->c.i : 0; // Value of right operand
#else
uint32_t l1 = c1 ? v1->c.i : 0; // Value of left operand
uint32_t l2 = c2 ? v2->c.i : 0; // Value of right operand
#endif
if (c2 && (op == '&' && l2 == -1)) {
  /* filter out NOP operations like x&-1... */
  vtop--;
}
```

Fig. 3.5. TCC code generator function with miscompiled 64-bit comparison.

Figure 3.5 shows an example where the `-1` literal is used in a comparison with a `uint64_t` variable. When compiling TCC with pnut (with the `HAVE_LONG_LONG` option disabled), both operands of the comparison are treated as 32-bit operands, resulting in the correct behavior. However, on the next compilation (this time with `HAVE_LONG_LONG` defined), `-1` is stored as a 32-bit value, but is extended (using `lexpand`) for the 64-bit comparison with `l2`. When using the original `lexpand`'s implementation, an incorrect comparison is generated using the zero-extended value: `l2_low == -1 && l2_high == 0`. If sign-extending the operand instead, the correct comparison is generated: `l2_low == -1 && l2_high == -1`.

It is difficult to catch these miscompilations, as they can be hidden by other miscompilations or correct themselves by chance after recompilation. Fortunately, outside of the `-1` examples, 64-bit types are used sparingly in TCC, with the few remaining large constant expressions appearing in the constant expression reduction code for cases that don't appear in TCC, meaning 64-bit types can be bootstrapped in three steps:

(1) The initial version of TCC (compiled by pnut) generates correct code for 64-bit arithmetic but cannot parse 64-bit literals.

(2) The second version of TCC has full support for 64-bit literals and generates correct code for 64-bit arithmetic. Constant folding of 64-bit expressions is still broken for some specific cases.

(3) The third version of TCC handles all 64-bit constant expressions correctly, generates correct code for 64-bit arithmetic and can parse literals.

3.4.2. Floating-Point Numbers in TCC

For floating-point numbers, TCC uses them in the following contexts [3]:

[3]found by looking for uses of `float`, `double`, `long double` and floating-point literals in the source code

```c
/* constant value */
typedef union CValue {
    long double ld;
    double d;
    float f;
#if HAVE_LONG_LONG
    uint64_t i;
#else
    uint32_t i; uint32_t i_padding;
#endif
    struct {
        int size;
        const void *data;
    } str;
    int tab[LDOUBLE_SIZE/4];
} CValue;

#if PTR_SIZE == 4
/* expand 64bit on stack in two ints */
static void lexpand(void)
{
    int v;
    v = vtop->r & (VT_VALMASK | VT_LVAL);
    if (v == VT_CONST) {    /* if operand is constant */
        vdup();             /* duplicate operand */
#ifdef HAVE_LONG_LONG
        vtop[0].c.i >>= 32; /* extract upper 32 bits */
#else
        if (vtop[0].c.i & 0x80000000 != 0) {
            /* Manually sign-extend */
            vtop[0].c.i = 0xFFFFFFFF;
        } else {
            vtop[0].c.i = 0;
        }
#endif
    }
    ... // rest of function unchanged
}
#endif
```

Fig. 3.6. TCC `lexpand` function that splits 64-bit operands into two 32-bit operands. Code active when `HAVE_LONG_LONG` is defined corresponds to TCC's original implementation, the other code is the modified version that works with `uint32_t CValue.i`.

```c
double strtod(const char *str, char **endptr) {
  long long res;
  if (strcmp(str, "0.0") == 0) {
    if (endptr) *endptr = (char *) str + 3;
    res = 0x00000000;
  } else if (strcmp(str, "1.0") == 0) {
    if (endptr) *endptr = (char *) str + 3;
    res = 0x3FF00000;
  } else if (strcmp(str, "4294967296.0") == 0) {
    if (endptr) *endptr = (char *) str + 12;
    res = 0x41F00000;
  } else {
    pnut_abort("stdtod: Unknown string: ");
  }
  res <<= 32; // Prevent constant folding
  return *((double *)&res);
}
```

Fig. 3.7. Lookup table-based implementation of the `strtod` function in pnut's C library.

- In the parser, to parse floating-point literals.
- In the code generator, for floating-point operations and constant expression evaluation. In particular, the negation of non-constant floating-point expressions depends on `-1.0` and `0.0`, creating a dependency on well-parsed floating-point literals.

The floating-point number parser uses two different parsing strategies depending on the base used, both requiring different levels of floating-point number support. For base 2 and base 16 literals, both the integer and fractional parts are parsed as 64-bit unsigned integers along with other information, before being converted to a double-precision floating-point number. For base 10 literals, the string is passed to the `strtof`, `strtod` or `strtold` functions depending on the suffix used (i.e. `f`, `d` or `ld` respectively), relying on the C library implementation to parse the string and convert it to a floating-point number. In the case of the C library used to bootstrap TCC, the `strto*` functions are implemented without the use of floating-point literals or expressions to avoiding a circular dependency. Since only the `0.0`, `1.0` and `4294967296.0` literals are used in TCC and in the C library, the functions can be implemented as a lookup table like in Figure 3.7. TCC does not use base 2 and base 16 floating-point literals, which can be left stubbed out.

The code generator uses floating-point operations and literals to implement constant expression reduction, including casts and arithmetic operations. TCC contains very few floating-point literals and even fewer floating-point expressions that can be reduced at compile time. It has one unfortunate problem, however, the implementation of floating-point negation relies on floating-point literals and the negation of floating-point numbers, creating a circular dependency as shown in Figure 3.8. The former is solved by implementing the `strtod` function as a lookup table, but the circular dependency of floating-point negation needs to be patched to break the cycle. This is done by hardcoding the bit pattern for `-0.0` in IEEE 754 format.

```c
case '-':
  next();
  unary();
  t = vtop->type.t & VT_BTYPE;
  if (is_float(t)) {
    // In IEEE negate(x) isn't subtract(0,x)
    , but rather subtract(-0, x)
    vpush(&vtop->type);

    if (t == VT_FLOAT)
      vtop->c.f = -1.0 * 0.0;
    else if (t == VT_DOUBLE)
      vtop->c.d = -1.0 * 0.0;
    else
      vtop->c.ld = -1.0 * 0.0;
  } else {
    vpushi(0);
  }
  vswap();
  gen_op('-');
  break;
```

Original implementation

```c
case '-':
  next();
  unary();
  t = vtop->type.t & VT_BTYPE;
  if (is_float(t)) {
    // In IEEE negate(x) isn't subtract(0,x),
     but rather subtract(-0, x)
    vpush(&vtop->type);
    vtop->c.i = 0x1; /* Stop constant fold */
    if (t == VT_FLOAT)
      vtop->c.i <<= 31; /* CValue.i */
    else if (t == VT_DOUBLE)
      vtop->c.i <<= 63; /* CValue.i */
    else
      vtop->c.ld = -1.0 * 0.0; /* Unused */
  } else {
    vpushi(0);
  }
  vswap();
  gen_op('-');
  break;
```

Patched implementation

Fig. 3.8. TCC function that handles negation of numeric expressions. Negation of floating-point numbers f is handled by evaluating $(-1.0 * 0.0) - f$, adding a dependency on floating-point literals and a dependency on floating-point negation to compute `-(1.0)`.

As a result, the constant expression evaluator does not impede the bootstrapping process, as the rest of the code generator does not use floating-point numbers, meaning that later stages of the bootstrap process will correctly compile the constant expression evaluator.

Because the parser and code generator both rely on some form of floating-point numbers, in addition to 64-bit integer types, bootstrapping floating-point numbers is done in multiple steps, each granting slightly more functionality to TCC until it can compile itself correctly:

(1) The initial version of TCC (compiled by pnut) generates correct code for floating-point operations, except for the negation operation. Floating-point literals are miscompiled because of missing 64-bit support.

(2) The second version of TCC generates correct code for floating-point operations, can parse literals and constant folding is now correct.
(3) Compile the non-patched version of TCC to get the final version of TCC.

3.4.3. Bootstrap Steps

Compiling TCC is done in multiple steps, each time using the previous version of TCC to compile the next version. Initially, `pnut-exe` compiles TCC and the libc together from the C source files, producing a statically linked executable. However, TCC expects the C standard library to be precompiled and present as a `.a` static library file, meaning that the TCC bootstrap is interleaved with the compilation of the libraries used by TCC. To bootstrap TCC, the following steps are performed:

(1) Compile the patched version of TCC with `pnut-exe` to get `tcc-pnut`. `Pnut-exe` produces an executable that is statically linked to the pnut libc, which is partly defined in C, and partly built-in to `pnut-exe`. `tcc-pnut` can generate the right instructions for floating-point operations (except negation) and 64-bit integers, but cannot parse floating-point literals and 64-bit integers, nor evaluate constant expressions involving them.
(2) Compile the pnut libc with `tcc-pnut`. TCC can compile inline assembly blocks, and so the built-in functions of `pnut-exe` can now be implemented in C, using inline assembly for system calls. The pnut libc is concatenated into a single C file, `libc.c`, which is then compiled with `tcc-pnut` to get a static library `libc.o`. The object file is then packaged into a static library file using the `tcc -ar` command, resulting in `libc.a`. Also, because code compiled with TCC is meant to be linked with the `libtcc1.c` runtime library, `libtcc1.c` is compiled to get `libtcc1.a`. This library is not used when bootstrapping TCC, but TCC expects it to be present when linking, so it must be generated.
(3) Compile the non-patched version of TCC with `tcc-pnut` to get `tcc-boot0`. This compilation uses `libtcc1.a` and `libc.a`. This version of `tcc-boot0` has partial support for floating-point numbers and 64-bit integers, being able to parse floating-point literals and generate the right instructions for floating-point operations and 64-bit integers, but not constant-folding them (since that code uses floating-point and 64-bit literals).
(4) Recompile the pnut libc with `tcc-boot0` to get a new `libc.o` and `libc.a`. Do the same for `libtcc1.c` to get `libtcc1.o` and `libtcc1.a`. This version of the libc and libtcc1 now have their literals properly compiled, but any constant expressions

involving floats or 64-bit integers are still potentially miscompiled. Fortunately, the libc and libtcc1 libraries contain no such expressions, so they are now fully functional.

(5) Compile the non-patched version of TCC with `tcc-boot0` to get `tcc-boot1`. `tcc-boot1` is linked to the final `libc.a` and `libtcc1.a`, and has now full support for floating-point numbers and 64-bit integers.

A fixed point is finally reached with `tcc-boot1`. Recompiling the pnut libc, `libtcc1.c` and TCC with `tcc-boot1` produces the exact same files, meaning the bootstrap process produced a reasonably functional TCC compiler and can stop here. Still, the version of TCC we've compiled is not the latest, nor is it built with a robust C library. The rest of the system, utilities and libraries also need to be bootstrapped to get a modern and functional system. These concerns, however, are out of the scope of this work as they are more related to the GNU/Linux system as a whole than to the bootstrapping of TCC on POSIX shell.

3.5. Reproducing the TCC Binary of Live-Bootstrap

The live-bootstrap project already provides a bootstrapping path to TCC, using MesCC and its C library to compile TCC. It can be useful to reproduce the same executable files as the live-bootstrap project to ensure that pnut gets to the same result. A different binary would indicate that the bootstrap process is not reproducible, likely due to a bug in pnut.

To reproduce the TCC binary of the live-bootstrap project, it is important that the environment matches the live-bootstrap environment as closely as possible; files must be in the same directories, commands with relative paths must be executed from the same directory, and the libc used must be the same. The version of TCC must also be the same, which is fortunately already the case, with the exception of a few pnut-specific patches that are needed only for the initial TCC compilation (`tcc-pnut`) and that are then reverted.

The live-bootstrap environment does not come with a preinstalled shell, and so `pnut-exe` was ported to M2-Planet, the C compiler that hosts Mes, so that `pnut-exe` can be bootstrapped from that environment. The compilation steps are very similar, and the same binary is produced when bootstrapping TCC from `pnut-exe` and from Mes, demonstrating the reproducibility of the TCC bootstrap process using pnut.

Chapter 4

Design of Pnut

In this section, we motivate the design choices used to make pnut a practical tool for bootstrapping. This includes the architecture of the compiler, the pnut C library and significant optimizations required to keep the execution time reasonable.

4.1. Architecture of Pnut

Pnut features two compiler backends: one targeting human-readable POSIX shell code and another generating machine code (currently only x86). Since both backends are used for bootstrapping, we distinguish between them with their respective names: `pnut-sh` and `pnut-exe`.

Because not every C construct can be compiled to human-readable POSIX shell, `pnut-sh` may only compile programs written in a specific subset of C. The main unsupported features are control flow mechanisms that don't have a direct equivalent, such as `goto` statements and `switch` fall-through. Additionally, not all C data types have a direct representation in shell; this includes floating-point numbers, unsigned integers and fixed-width integers. These features are not essential for the implementation of a compiler, but must be supported by any compiler that hopes to support TCC.

This gap between POSIX shell and C99 is bridged using `pnut-exe`, which supports most C99 features and can compile TCC with a few patches. `Pnut-exe` shares the same frontend as `pnut-sh` and is implemented using the C subset supported by `pnut-sh`. This means `pnut-exe` can be compiled by `pnut-sh` and hosted by a POSIX-compliant shell.

4.1.1. Main Challenges

After the generation of human-readable shell code, the size and execution time of the `pnut-sh.sh` and `pnut-exe.sh` scripts are real concerns for the usability of pnut for reproducible builds.

A rule of thumb often used for compilers is that an optimization should pay for itself, meaning that the faster compiler obtained from the better code generation offsets the increased compilation time from the optimization. Most compilers make some effort to produce reasonably fast code, with some of them expending considerable time applying optimization passes to produce the fastest code possible.

However, this principle can be challenging to apply to pnut, as the bootstrapping steps that run on a shell take a disproportionate amount of time compared to the steps that follow. And because the code that is generated has to match the C code as much as possible, generating faster shell code is hardly an option. As a consequence, optimizations that would improve the execution speed, such as inlining, dead code elimination, loop unrolling and constant folding, are not performed by pnut. This implies that the primary way to reduce the time it takes to get to the executable `pnut-exe` is to generate code more efficiently, rather than generating faster code.

As shown in Chapter 5, memory usage is strongly correlated to execution time on certain shells. This means that for pnut to be usable across all shells, memory must be treated as a precious resource. A common approach for compilers used in low-resource environments is for them to be one-pass, a technique we adopted for pnut. One-pass compilers process their input once, processing a unit of code (top-level declarations for pnut) before outputting its result and repeating the process for the next units. By doing so, only the information relevant to the declaration is retained in memory, with a small state being maintained between declarations to keep track of the functions, global variables, type declarations (unions, structs, typedefs) and macro declarations that comprise the global scope.

4.1.2. Minimal Variants of Pnut

`Pnut-exe` serves two objectives with conflicting requirements: support enough of the C language to compile TCC, while being small and fast to bootstrap from `pnut-sh.sh`. To meet these two goals, `pnut-exe` comes in two variants: a complete variant, with support for all the C features needed to compile TCC as well as more command line options and usability improvements, and a minimal variant, providing only the features required to bootstrap `pnut-exe`. Because `pnut-sh` and `pnut-exe` share the same compiler frontend, `pnut-sh.sh` also comes in a minimal variant, removing unused features to keep the size of the `pnut-sh.sh` seed script small and easy to review. Table 4.1 shows the size (in lines of code) of both variants of `pnut-sh` and `pnut-exe`, while Table 4.2 shows the compilation time improvements when using the minimal variants.

The minimal variant is derived from the complete pnut source code using preprocessor directives to disable features as needed. Doing so keeps the source code shared between

Table 4.1. Size of minimal and complete variants of pnut.

pnut version	C source		Shell code	
	Complete	Minimal	Complete	Minimal
`pnut-sh.c`	5232	4263	7630	6211
`pnut-exe.c`	5029	4089	7024	5687

Table 4.2. Time to bootstrap the minimal and complete variants of `pnut-exe.c` per shell.

Shell	`pnut-sh.sh pnut-exe.c`		`pnut-exe.sh pnut-exe.c`	
	Complete	Minimal	Complete	Minimal
ksh	22.8 s	18.9 s (0.83 ×)	33.6 s	25.6 s (0.76 ×)
dash	59.3 s	43.2 s (0.73 ×)	172.3 s	107.7 s (0.63 ×)
bash	58.4 s	46.9 s (0.80 ×)	81.6 s	63.5 s (0.78 ×)
yash	57.4 s	47.9 s (0.83 ×)	86.3 s	66.5 s (0.77 ×)
zsh	773.0 s	532.7 s (0.69 ×)	817.7 s	562.4 s (0.69 ×)

the complete and minimal variants of the compiler, simplifying development as well as the auditing process.

Unless specified otherwise, all references to `pnut-sh` and `pnut-exe` in this chapter refer to their minimal variants, as those are the ones used for bootstrapping `pnut-exe` from shell. Once the minimal executable `pnut-exe` is bootstrapped, the complete version of `pnut-exe` can be compiled quickly to obtain the fully-featured compiler.

4.1.3. Compilation Pipeline

Some one-pass compilers emit code directly as tokens are read, avoiding the creation of an abstract syntax tree. This approach strongly couples the code generator to the frontend, which complicates development when multiple compiler backends are used. As such, pnut uses an abstract syntax tree as its intermediate representation.

Traditionally, compilers are used through a compiler driver, which sequences the preprocessing, code generation, assembly and linking as each step is done using a separate tool. This simplifies each individual tool, as each is tasked with only a portion of the whole pipeline, and allows tools to be combined and reused for different purposes. However, the overall complexity is higher as the tools need to communicate with each other through a text-based format. To avoid these downsides, pnut combines all four tools in one, preprocessing, compiling and outputting a finished executable all in one go.

4.1.4. Reader

For the reader, the entire file is read character by character, with no support for trigraphs or escaped newlines (\ followed by a newline). Simply reading the whole file at once would be impractical, as the memory usage would immediately overwhelm certain shells. To support `#include` directives, the reader maintains a stack of open files so it can start reading new files as they are included, and resume reading the parent file when it is done. The include stack also keeps track of the file path to compute the path of relative includes. For debugging, with the right compilation options, the reader can keep track of the line and column number so errors can be tagged with a location. This state is also part of the include stack.

4.1.5. Lexer

Next, the lexer consumes each character and produces a stream of tokens representing C language keywords, identifiers, operators and literals. Keyword, identifier and string literal tokens are interned so the underlying string can be shared between identical tokens, allowing constant-time comparison using pointer equality. The intern table also serves as the symbol table, storing metadata associated with each identifier, such as its kind (keyword, macro, typedef or other identifier), as well as metadata used for macros and typedefs. Doing so avoids the need for separate tables for typedefs and macros, making the implementation smaller and avoiding redundant table lookups.

Because the preprocessor is an important part of the C language, the lexer recognizes the following constructs, preprocessing the file as it is read:

- Object macros such as `#define FOO 123`
- Function-like macros such as `#define BAR(X) X + FOO`
- Conditional groups: `#if`, `#ifdef`, `#else`, `#elif` and `#endif`
- Diagnostic macros: `#warning` and `#error`
- Predefined macros: `__FILE__` and `__LINE__`
- System and user includes: `#include <stdio.h>` and `#include "foo.h"`

We note that macros can expand to other macros, or take macros as parameters, both of which result in further expansion, with the caveat that self-referential macros expand indefinitely. This departure from the C language simplifies the expansion logic and eliminates the need to double scan the list of macro tokens to prevent self-referential macros from expanding indefinitely. As a result, the lexer blindly expands macros as they are read, performing their expansion and recursively expanding any macro that is encountered. Self-referencing macros are detected by hitting the maximum expansion depth, which raises an error.

Because preprocessor macros are global, they are kept in the symbol table, alongside their arity and their list of tokens, for the entire execution (unless `#undef` is used). An unfortunate consequence of this design is that it prevents the symbol table from ever being cleared between declarations. The symbol table is one of the few data structures of pnut that never shrinks. In practice, however, most symbols in pnut are the same, and thus the table size remains relatively stable.

4.1.6. Parser

The parser is a simple recursive descent parser for the C subset supported by pnut. Like the reader and lexer, the parser is hand-written, so its implementation can be tailored to our specific use case. No external tools, such as parser generators, were used, as using machine-generated source files is to be avoided in reproducible build processes, especially when the generated code is not human-readable.

The parser advances one top-level declaration at a time, producing an abstract syntax tree (AST) for it and passing it to the code generator. The code generator can then process the AST and output the code for that declaration before returning to the parser to parse the next top-level declaration. AST objects end up taking a significant portion of the memory used by pnut. Unfortunately, as things are currently implemented, some AST nodes are kept alive until the end of the compilation, preventing the reuse of the memory allocated for them.

The grammar of the C language [28, Appendix A] is not quite LL(1), since typedef statements can dynamically extend the grammar to include new type identifiers, and certain production rules such as abstract and direct declarators share a common prefix which prevents the parser from distinguishing the two from a single token. It is close enough, however, that the parser can be extended to parse a larger superset of C that is LL(1). As a result, the parser may accept programs that are not valid C programs, but many of these are then rejected by the code generator. Either way, pnut is meant to compile TCC, which is a known valid C program, meaning this deviation from the C grammar is acceptable.

Solving the typedef problem is achieved using the lexer hack, where the parser informs the lexer of new typedefs as they are encountered by changing the symbol kind of identifiers so they are recognized as type identifiers instead of regular identifiers.

4.1.7. POSIX Shell Code Generator

As an interpreted language, POSIX shell offers many features that simplify its use as a compilation target. One such feature is that forward references to functions and variables that are not yet defined or declared are allowed. This simplifies the implementation of a one-pass compiler as it eliminates the need for many fixups and indirections. As a result, the challenges in implementing a POSIX shell code generator come from other constraints, such as generating human-readable code. First, to be human-readable, the shell code must be appropriately indented. Second, shims that are introduced by the shell code generation must be seamlessly integrated with the rest of the code.

`Pnut-sh` compiles the statements of functions one after the other, each mapping to a few lines of shell code. Constructing each line of shell code is heavily dependent on string concatenation, which can result in quadratic time and memory complexity if implemented naively. To solve this problem, `pnut-sh` uses a tree-like data structure to represent string conversions and concatenation, similar to the `rope` data structure. Generally, conversions of strings, such as escaping or formatting numbers (decimal, hexadecimal and octal), are best delayed as the conversion would expand their size. To further reduce memory usage, the data structure also differentiates between immutable strings, which come from the string intern pool and string literals, and mutable strings, which must be copied.

A second data structure keeps track of the lines that have been generated and ensures they are properly indented. At specific points in the generated code, placeholders can be inserted to allow the insertion of the required shims once the function declaration has been fully processed. These shims are used to implement the function prologue and epilogue, which appear at the beginning and end of the function, but also wherever the function returns early. The data structure also allows lines to be removed, copied, and merged, and can be queried to count the number of lines between two points, useful for generating syntactically valid shell code as empty blocks are disallowed.

Once the code for a function is compiled and the prologue and epilogue are inserted, the structure is traversed in order and each line is written directly to the output, avoiding the allocation of temporary buffers. Both data structures are stored in preallocated buffers that are reset and reused between each top-level declaration. Reusing these buffers reduces total memory usage by more than half when compiling `pnut-exe.c`, which has a significant impact on the performance on the slowest shell implementations. Table 4.3 and Table 4.4 summarize the memory usage and compilation time improvements brought by the one-pass compilation strategy when compiling `pnut-sh.c` and `pnut-exe.c`. To simulate a multi-pass compiler, the code is accumulated until the end of the compilation, so the code buffers are never emptied.

To avoid generating dead code, special care was taken to include only the parts of the runtime library referenced by the shell code. To do so, calls to runtime library functions are tracked. This reduces the amount of code generated, which in turn reduces compilation time and bloat of the generated scripts.

Most lines of the runtime library are used for I/O functions such as `open`, `read`, `write` and `printf`. A reasonably complete C `printf` implementation takes a few hundred lines of

Table 4.3. Memory reduction achieved through one-pass code generation when compiling `pnut-sh.c` and `pnut-exe.c` (i386 Linux) using `pnut-sh.sh`. Memory usage is measured by counting the number of shell variables that are written to during compilation.

`pnut-sh` version	`pnut-sh.c`	`pnut-exe.c`
One-pass	151087	122553
Multi-pass	340967	292780
Ratio	2.26 ×	2.39 ×

Table 4.4. Performance impact of the memory reduction from the one-pass code generation when compiling `pnut-sh.c` and `pnut-exe.c` (i386 Linux) using `pnut-sh.sh`.

Shell	One-pass (s)		Multi-pass (s)	
	pnut-sh.c	pnut-exe.c	pnut-sh.c	pnut-exe.c
ksh	20.6 s	18.9 s	21.5 s (1.04 ×)	18.7 s (0.99 ×)
dash	54.5 s	43.2 s	150.2 s (2.76 ×)	123.2 s (2.85 ×)
bash	52.0 s	46.9 s	52.6 s (1.01 ×)	46.4 s (0.99 ×)
yash	53.4 s	47.9 s	52.9 s (0.99 ×)	47.4 s (0.99 ×)
zsh	690.0 s	532.7 s	3441.5 s (4.99 ×)	1911.0 s (3.59 ×)

shell code, which need to be included with every program that uses `printf`. However, since POSIX shell already includes a `printf` built-in function, it is often possible to reuse the shell's built-in `printf` as long as the format string is known at compile time. By doing so, the conversion of the format string to a C string can be avoided, and the string formatting is done directly by the shell. The result is a shorter, as the `printf` implementation is not needed, and faster shell script.

Unfortunately, the representation of C strings in shell is incompatible with the shell's built-in `printf`. This means format strings with `%s` cannot be passed through directly to the shell's `printf` as the C strings would first need to be converted to shell strings. As such, `pnut-sh` splits `printf` calls so that the parts of the format string that map one-to-one to the built-in `printf` can be passed directly, and replaces `%s` with calls to a function that outputs the characters of the C string one by one.

This optimization is one of the few shell code optimizations that pays for itself. The code that produces the `pnut-sh` runtime library is the best example of this, as the implementation is more or less a series of `printf` calls with known format strings (See Figure 4.1). As a result, the built-in `printf` can be used, which avoids the parsing of the format string at runtime and generates more idiomatic code. Doing so also saves many object allocations, as

```
bool runtime_make_argv_defined = 0;
void runtime_make_argv() {
  if (runtime_make_argv_defined) return;
  runtime_make_argv_defined = 1;
  runtime_malloc(); // Include malloc function
  runtime_unpack_string_to_buf(); // Include string unpacking function
  printf("make_argv() {\n");
  printf("  __argc=$1; shift;\n");
  printf("  _malloc __argv $__argc\n");
  printf("  __argv_ptr=$__argv\n\n");
  printf("  while [ $# -ge 1 ]; do\n");
  printf("    _malloc _$__argv_ptr $((${#1} + 1))\n");
  printf("    unpack_string_to_buf \"$1\" $((_$__argv_ptr)) 1\n");
  printf("    : $((__argv_ptr += 1))\n");
  printf("    shift\n");
  printf("  done\n");
  printf("}\n\n");
}
```

Fig. 4.1. Function from `pnut-sh` outputting a runtime library function.

the conversion of the format strings into C strings allocates one word per character, and the runtime library is made up of more than 10000 characters that would stay in memory until the end of the execution.

4.1.8. Machine Code Generator

`Pnut-exe` generates machine code directly from the AST produced by the parser, outputting a position-independent executable that is statically linked to the built-in C library. `Pnut-exe` supports 32-bit (i386) and 64-bit (amd64) x86 architectures, and the ELF and Mach-O executable formats for compatibility with Linux and macOS. The ELF and Mach-O files generated are very minimal, containing the smallest header possible, followed directly by the binary code in a single `.text` section. Constant data, such as string literals, are located directly in the `.text` section, ensuring they are read-only. Statically allocated memory is allocated either on the stack or in a memory-mapped region (configurable with build options), and initialized at runtime by the program. Since the executable is position-independent and only contains a single section, the code to generate these executables is minimal. `Pnut-exe`'s ELF code is approximately 100 lines, making it easy to audit and understand. For comparison, TCC's `elf.h` header file is over 3000 lines long, supporting a wide range of features that are not needed by `pnut-exe`.

4.1.8.1. Stack Machine and Calling Convention. To simplify the code generation, `pnut-exe` generates code for a stack machine, where the operands of every expression are kept on the stack. The stack is also used to pass the function call arguments, using the `cdecl` calling convention. Stack machine code generators are generally easier to program than their register machine counterparts, as they don't involve register allocation. The encoding of instructions is also simplified, as the same three registers are always used (two for performing binary operations, one temporary for copy operations), removing the need to support the encoding of instructions with all registers and other addressing modes.

A potential downside of using a stack machine is increased code size. To evaluate the performance impact of reducing code size, we compare the time to compile `pnut-exe.c` by using shorter instruction encoding where possible in the x86 instruction encoder. This includes encoding small 32-bit immediates and displacements as sign-extended bytes, using `xor` to clear registers and removing redundant `mov` and `add` instructions. Table 4.6 shows the limited performance impact of further code size reductions on the performance of `pnut-exe.sh`. This is due to the one-pass code generator already using relatively little memory compared to the rest of the compiler. Because the performance impact of more compact instruction encoding is minimal and limited to a few shells, and to keep the instruction encoding logic easy to audit and understand, we decided not to implement these optimizations. This result also suggests that using a stack machine architecture is beneficial for code generation simplicity, even if it uses more memory.

4.1.8.2. One-Pass Code Generation. Like `pnut-sh`, `pnut-exe` generates code in one pass, processing each top-level function declaration before outputting the machine code for it, and reusing the same small code buffer between declarations. This cuts down in half the memory usage when compiling `pnut-exe.c`, which has a significant impact on the performance on the slowest shell implementations. Table 4.5 and Table 4.6 summarize the memory usage and compilation time improvements brought by the one-pass compilation strategy when compiling `pnut-exe.c`. Like for `pnut-sh`, a multi-pass compiler is simulated by accumulating the generated code until the end.

However, the one-pass generation of machine code introduces additional complexity, since machine code inherently requires local fixups that can only be performed on code that is still in memory. Unlike with more compact instruction encoding that we decided not to use, the improvements in compile times are well worth the increased complexity. The difficulties come in 2 variants:

(1) Forward jumps to functions that are not yet defined.
(2) The size of specific structures is needed before they are known.

Table 4.5. Memory reduction achieved through one-pass code generation and compact instruction encoding when compiling `pnut-exe.c` using `pnut-exe.sh` (i386 Linux). Memory usage is measured by counting the number of shell variables that are written to during compilation.

`pnut-exe` version	Default encoding	Compact encoding	Ratio
One-pass	119398	118163	0.99 ×
Multi-pass	259259	231200	0.89 ×
Ratio	2.17 ×	1.96 ×	

Table 4.6. Performance impact of the one-pass code generation and compact instruction encoding when compiling `pnut-exe.c` with `pnut-exe.sh` (i386 Linux). The one-pass with default encoding column corresponds to the implementation used in `pnut-exe`.

Shell	One-pass (s)		Multi-pass (s)	
	Default encoding	Compact encoding	Default encoding	Compact encoding
ksh	25.6 s	25.8 s (1.01 ×)	26.3 s (1.03 ×)	25.8 s (1.01 ×)
dash	107.7 s	107.1 s (0.99 ×)	399.8 s (3.71 ×)	322.3 s (2.99 ×)
bash	63.5 s	62.9 s (0.99 ×)	61.8 s (0.97 ×)	61.6 s (0.97 ×)
yash	66.5 s	66.7 s (1.00 ×)	66.3 s (1.00 ×)	65.9 s (0.99 ×)
zsh	562.4 s	546.8 s (0.97 ×)	1753.6 s (3.12 ×)	1300.9 s (2.31 ×)

Forward jumps typically appear in calls to functions that are defined later in the code and when initializing global variables, since their initialization is interspersed with the rest of the code. This interspersing is done to avoid having to keep the global variable initializers until the very end, but it also means that the code of the initializers must be chained with forward jumps, as shown in Figure 4.2.

Allocating enough space for global variables can also be a challenge since the total space they occupy is unknown to the compiler, but their allocation must be done before any initialization code is executed.

Finally, ELF and Mach-O executable file headers require the length of the executable to be present at the beginning of the file. Figure 4.3 shows a 32-bit ELF header as generated by `pnut-exe`, showing that the `p_filesz` and `p_memsz` fields can only be computed at the end of the program. This constraint, if enforced strictly by the operating system, makes it more or less impossible to have a true one-pass compiler without precomputing the program length. Fortunately, these problems can be solved with a few tricks.

For forward jumps to functions, the code generator maintains a global offset table (GOT) that stores the address of all functions of the program. The GOT is initialized at program startup (and not by the program loader) and is used to resolve forward jumps to functions that have not yet been defined. When the function is declared (but not necessarily defined), the code generator assigns an entry to store the address of the function that will be initialized with the address of the function. Forward function calls thus load the address of the target function from the table and call it indirectly. This indirection makes forward function calls more expensive, but the difference is negligible compared to the time saved from running `pnut-exe.sh` with the one-pass code generator. Additionally, to mitigate the cost of forward calls, built-in functions are located at the beginning of the program, so calls to these functions use the faster direct call instruction.

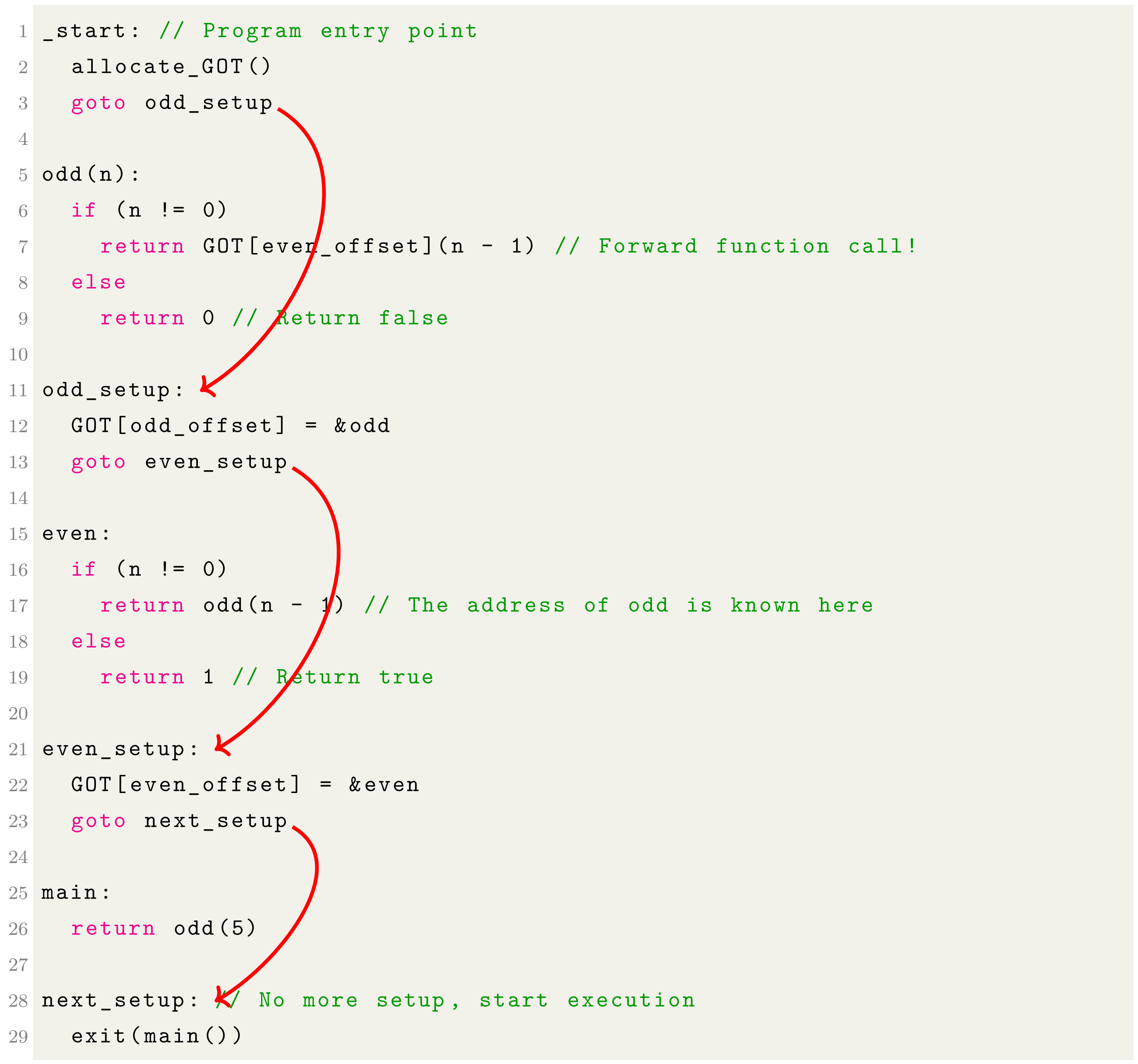
To be able to output the code after a function definition, the destination of all jumps must be known. Fortunately, local jumps inside a function are all resolved at the end of the function, leaving the label of the next initialization block as the only unresolved jump destination. Because the function definition is followed by the initialization of its global offset table entry, the setup label is temporarily resolved. At this point, all labels are resolved and the code can be written out. In Figure 4.2, the points where all labels are resolved are after lines 12 and 22.

For the size of global variables, it is assumed to be under a hardcoded limit. Because global variables are allocated in a memory-mapped region, overestimating the size of the global variables has no measurable impact on the performance, as additional memory pages are left untouched.

For the program size in the executable file headers, on platforms such as Linux, the size in the ELF header does not need to match the program code size. As long as the declared size is greater or equal than the actual size, the program loads without issue. Unfortunately, this solution is not compatible with all platforms, as is the case with macOS, requiring an additional code generation pass to compute the length of the machine code.

4.1.9. Bytecode Interpreter

Since the runtime performance of the machine code is not a constraint on the design of pnut, an alternative to implementing a native code generator could be to use a bytecode interpreter and have `pnut-exe` target its bytecode instead. With the proper bytecode, this could facilitate the implementation of a one-pass compiler, as resolving the forward references could be left to the interpreter (where doing multiple passes is acceptable). Bytecode is also portable, as only the virtual machine is tailored to a specific architecture, more compact, as higher-level instructions can be used, and potentially easier to debug.



```
_start: // Program entry point
  allocate_GOT()
  goto odd_setup

odd(n):
  if (n != 0)
    return GOT[even_offset](n - 1) // Forward function call!
  else
    return 0 // Return false

odd_setup:
  GOT[odd_offset] = &odd
  goto even_setup

even:
  if (n != 0)
    return odd(n - 1) // The address of odd is known here
  else
    return 1 // Return true

even_setup:
  GOT[even_offset] = &even
  goto next_setup

main:
  return odd(5)

next_setup: // No more setup, start execution
  exit(main())
```

Fig. 4.2. Layout of machine code generated by `pnut-exe` in pseudo-assembly for the mutually recursive `odd` and `even` functions. Arrows indicate the control flow during initialization, with top-level functions having their GOT (Global Offset Table) entry initialized. The `odd_setup`, `even_setup` and `next_setup` labels are used to chain the initialization blocks.

We chose not to use this method in `pnut-exe` for several reasons, the primary reason being that bootstrapping the bytecode interpreter would require hardcoding its machine code in some form, which goes against the goal of auditability. This problem could be alleviated by having functions to encode instructions and to generate files in an executable format, making the interpreter implementation easier to review, but that is not far from having a complete native code generator. The end result would likely be a larger and more complex compiler,

```
ehdr:                         ; Elf32_Ehdr
    db    0x7F, "ELF"         ;   e_ident
    db    1, 1, 1, 0          ;   e_ident (cont)
times 8 db      0
    dw    2                   ;   e_type
    dw    3                   ;   e_machine
    dd    1                   ;   e_version
    dd    _start              ;   e_entry (entry_point_address)
    dd    phdr - $$           ;   e_phoff
    dd    0, 0                ;   e_shoff, e_flags
    dw    ehdrsize            ;   e_ehsize
    dw    phdrsize            ;   e_phentsize
    dw    1, 0, 0, 0          ;   e_phnum, e_shentsize, e_shnum, e_shstrndx

ehdrsize  equ    $ - ehdr     ; Compute header size

phdr:                         ; Elf32_Phdr
    dd    1, 0                ;   p_type, p_offset
    dd    $$                  ;   p_vaddr
    dd    $$                  ;   p_paddr
    dd    filesize            ;   p_filesz (program_size)
    dd    filesize            ;   p_memsz  (program_size)
    dd    5, 0x1000           ;   p_flags, p_align

phdrsize  equ    $ - phdr     ; Compute program header size

_start:
  ... program code...

filesize  equ    $ - $$       ; Compute program size
```

Fig. 4.3. 32-bit ELF header generated by `pnut-exe` in NASM assembly.

the exact opposite of what we want. Implementing the bytecode interpreter in POSIX shell (or in C compiled with `pnut-sh`) may seem like a solution to this problem. Unfortunately, it is not a feasible solution as every step until TCC recompiles itself would execute on the shell, which would be too slow to be usable.

Other reasons are that the portability of the bytecode is not particularly useful since porting the interpreter to a new platform involves a similar amount of work to extending the native code generator for that platform. Finally, regarding debugging, the main advantage a bytecode interpreter has over an executable comes from its higher-level instructions. However, debugging a bytecode program requires tooling that would need to be built, as opposed to regular executables that are compatible with off-the-shelf debuggers.

4.2. Built-in Library

The C language is not very useful without its standard library. Pnut provides a minimal implementation to bootstrap the compilation environment, `pnut-exe` and TCC. Many of the C standard library functions cannot be implemented in plain C. In modern C compilers, this problem is usually solved with fragments of inline assembly that perform calls to the operating system. Pnut does not support inline assembly and instead detects the use of certain functions and defines them as needed. We call the set of such functions the *built-in library*. The set of built-in functions varies between `pnut-sh` and `pnut-exe`, as the shell and operating systems provide different capabilities, and because `pnut-sh`, `pnut-exe` and TCC all require a distinct subset of the C standard library.

`Pnut-sh` provides the following functions: `putchar`, `getchar`, `exit`, `malloc`, `free`, `printf`, `fopen`, `fclose`, `fgetc`, `open` and `close`, `read` and `write`. Some of these functions, such as `fopen` and `printf`, could be implemented in C from lower-level primitives. Having them as built-in makes `pnut-sh` self-contained, helping with the distribution and auditing of `pnut-sh.sh`. Built-in functions also don't need to be parsed by the compiler, reducing compilation time as the built-in library is around 1000 lines of shell code, and can be included as needed to keep the shell scripts free of dead code. The latter would require dead code elimination, which cannot be implemented in a one-pass compiler. Built-in functions can also be manually optimized, for example, by using a separate namespace for local shell variables to avoid the overhead of `let` and `endlet`. However, built-in functions bloat the `pnut-sh.sh` script by appearing twice: once as shell code and once embedded in the compiler so it can reproduce the functions. This duplication could potentially be solved by dynamically producing the built-in library during the program initialization, but this would require the use of `eval`, which we consider worse than the duplication.

`Pnut-exe` provides the following functions: `exit`, `read`, `write`, `open`, `close`, `lseek`, `unlink`, `mkdir`, `chmod` and `access`. The last three functions, `mkdir`, `chmod` and `access`, are specifically for bintools. TCC utilizes many more functions from the standard library, but many are never called when compiling TCC, meaning they can be left unimplemented and replaced by stubs. Other functions, such as `malloc`, `free` and `printf`, can be implemented directly in C, and are part of the pnut C standard library.

Because TCC supports inline assembly, once TCC is compiled, we can reuse off-the-shelf libc implementations that are more complete and performant to obtain a more complete version of TCC.

Chapter 5

Compiling C to POSIX Shell

This chapter covers the mapping from the C language to POSIX shell used in `pnut-sh`. Given pnut's goal to have easily auditable generated code, the following design principles guided our choices when multiple implementations were possible:

- Preserve the general structure of the C source code.
- Generate regular shell code and avoid special cases.
- Avoid the use of hardcoded constants and magic numbers in the shell code.
- Avoid the use of the shell `eval` command and other kinds of dynamic code evaluation, both for readability and efficient safety auditing.
- Prioritize readability over performance, but don't ignore performance.

These constraints and the low-level nature of the C language make it difficult to utilize certain features of the shell language. For example, the shell string processing utilities are too high-level to be useful for the low-level operations on C strings. The opposite is also true, where some C constructs are difficult to map to shell. As a result, certain C features are not supported, including:

- `goto` and `switch` case fall-through: The shell doesn't have a direct way to implement this kind of control flow, and supporting it would require transformations that would diminish its readability.
- The *address of* (`&`) operator on local variables: Supporting this would prevent the direct mapping of C local variables to shell variables.
- Floating-point numbers: Floating-point numbers are not supported natively by the shell and would require a large amount of code to implement fully (recall that making use of external programs such as the Unix `bc` is forbidden by our design constraints).

These features are not necessary for a large class of programs and, in particular, were not used for writing `pnut-sh.c` and `pnut-exe.c`. We note that these limitations stem from the requirement for readability and auditability of the shell code. The `pnut-exe` compiler, which targets machine code, is not subject to these limitations and is thus suitable for building TCC, whose source code uses these features. That being said, `pnut-sh.sh` supports a comfortable subset of the C language, including all signed arithmetic operators, structures, enums, pointers, pointer arithmetic and C preprocessor directives.

Additionally, as some of the implementation choices taken are at the cost of performance, we evaluate their impact in comparison to the other options. The following table gives the time taken by the minimal variants of `pnut-sh.sh` and `pnut-exe.sh` (i386 Linux) to compile `pnut-exe.c`, two important steps in the bootstrapping process, on different shells with default options. These measurements will serve as the baseline when comparing different code generation approaches and are shown in Table 5.1.

Table 5.1. Baseline execution times of `pnut-sh.sh` and `pnut-exe.sh` compiling `pnut-exe.c`.

Shell	`pnut-sh.sh pnut-exe.c`	`pnut-exe.sh pnut-exe.c`
ksh	18.9 s	25.6 s
dash	43.2 s	107.7 s
bash	46.9 s	63.5 s
yash	47.9 s	66.5 s
zsh	532.7 s	562.4 s

For reference, when we measure performance to support our claims and choices, it is done in the hardware/software environment which corresponds to a typical development environment with recent versions of various shells. The precise specifications are shown in Figure 5.1.

OS:	Ubuntu 22.04.5 with Linux 6.8.0-65
Processor:	AMD Ryzen 9 5900X @ 4.95GHz
RAM:	128GiB
shells:	`bash` 5.1.4(1)-release, `dash` 0.5.11, `ksh` 93u+m/1.0.0-beta.2, `yash` 2.51, `zsh` 5.8.1

Fig. 5.1. Hardware/software environment used for performance measurements.

5.1. Shell Variables and Functions

Because all shell variables live in the same namespace, it is essential to use a naming scheme that prevents conflicts between the internal variables used by `pnut-sh` and the variables of the compiled program. To this end, the underscore character serves as a prefix to create separate namespaces for the variables of the compiled program. Because local variables are most common, they are assigned the empty namespace and have the same name as in the C code. Global variables are prefixed with one underscore to help distinguish them from local variables, and internal variables are prefixed with two underscores. This naming scheme is short and easily recognizable and is used throughout the shell code generated by `pnut-sh`. This prevents the use of C variables starting with an underscore, but it is an acceptable compromise as it improves readability to preserve the same name.

One kind of internal variable is temporary variables created to store intermediate results. These use the `__t` prefix and are numbered starting from 1. Using the internal variable namespace makes it clear that these variables are not present in the C code and were generated by the compiler.

In addition, certain variables such as `IFS` and `PATH` have a special meaning in the shell. These variables are not allowed by `pnut-sh`. While `pnut-sh` could assign different variable names to these special variables, that would make the generated shell code less readable and these variables can easily be avoided by the programmer. An exception to this rule is the `argv` variable, which has a special meaning in `zsh` and is commonly used in C for the command line arguments received by `main`. This variable is mapped to `argv_` in the generated shell code, and the C variable `argv_` is forbidden to ensure that there are no naming conflicts.

All function names are prefixed with an underscore with no restriction on the function name. This leaves the namespace with no prefix available for the runtime library functions, preventing conflicts with the shell's built-in functions and utilities.

5.2. Calling Convention

A major difference between POSIX shell and C is the treatment of functions. POSIX shell procedures cannot declare local variables, as shell variables are globally scoped, and they cannot return a result value. We will nevertheless refer to them as shell *functions* as this is the commonly used term.

5.2.1. Returning from Functions

One way to return a value from a function is to assign the value to a global variable, which can then be read by the function's caller. Using the fact that shell allows assigning to variables with computed names using string and arithmetic expansion, we choose to have functions take as their first argument the name of the variable that receives the result, the *return variable*. We believe this style reads easily as all function calls have the same structure. Also, since the results of C functions are often assigned to variables, this choice makes it

```
__SP=0
let() { # $1: variable name, $2: optional value
  : $((__SP += 1)) $((__$__SP=$1)) # Push
  : $(($1=$2+0))                   # Init
}

endlet() { # $1: return variable
           # $2...: function local variables
  __ret=$1 # Don't overwrite return value
  : $((__tmp = $__ret))
  while [ $# -ge 2 ]; do
    : $(($2 = __$__SP)) $((__SP -= 1)); # Pop
    shift;
  done
  : $(($__ret=__tmp)) # Restore return value
}
```

Fig. 5.2. Implementation of `let` and `endlet`.

possible for a function to assign the result directly where the caller wants the result to be. Concretely, the C function call `x = f(y, 1)` is mapped to the shell code `_f x $y 1`.

In the cases where the result of a function is ignored, the internal variable `__` is used as the return variable as a convention. The C program cannot access this shell variable, and so any value written to it is effectively discarded. The presence of `__` in a function call is a reminder that the result of the function is ignored.

5.2.2. Local Variables

We use the term *local variable* to refer to a programmer-declared local variable or parameter, or a compiler-introduced temporary. Since POSIX shell offers only globally scoped variables, local scoping of variables is achieved with a *shallow binding* [4] approach. For this, we use a callee-save calling convention. Every function begins by saving its local variables to a stack so that the function is free to use them. The variables get restored from the stack when the function returns. Unlike the traditional C stack implementation, this stack is used solely for saving local variables and is not used for control flow, meaning no stack space is used when there are no local variables.

To make the bookkeeping of local variables easy to read, it is abstracted using the `let` and `endlet` functions with Figure 5.2 showing their implementation.

The `let` function takes the name of a local variable as its first argument and optionally takes the initial value as the second argument. It saves the value of the variable to the stack

```c
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

C source code

```sh
_fib() { let n $2
  let __t1; let __t2
  if [ $n -lt 2 ] ; then
    : $(($1 = n))
  else
    _fib __t1 $((n - 1))
    _fib __t2 $((n - 2))
    : $(($1 = __t1 + __t2))
  fi
  endlet $1 __t2 __t1 n
}
```

Using `let` and `endlet`

```sh
_fib() { set $@ $n $__t1 $__t2
  n=$2
  if [ $n -lt 2 ] ; then
    : $(($1 = n))
  else
    _fib __t1 $((n - 1))
    _fib __t2 $((n - 2))
    : $(($1 = __t1 + __t2))
  fi
  : $((__tmp=$1)) $((n=$3)) $((__t1=$4)) $((__t2=$5)) $(($1=__tmp))
}
```

Using `set`

Fig. 5.3. Comparing `let`/`endlet` with `set` for Fibonacci.

and assigns its initial value or 0 if the initial value is not specified. The `endlet` function takes the name of the return variable as its first argument, followed by the names of the variables to restore. It ensures that the return variable is not overwritten in case a local variable of the callee has the same name as the return variable specified by the caller.

Calls to `let` appear at the head of function bodies, and calls to `endlet` appear last. These statements always come together, but `endlet` can undo the effect of multiple `let`s and may appear in the middle of a function returning early.

5.2.2.1. Positional Parameters for Local Storage. This bookkeeping has a cost, since all functions, no matter how small, require it as they all have the return variable parameter

Table 5.2. Execution times of `pnut-sh.sh` and `pnut-exe.sh` *not* using positional parameters for constant parameters.

Shell	`pnut-sh.sh pnut-exe.c`	`pnut-exe.sh pnut-exe.c`
ksh	24.5 s (1.30 ×)	36.8 s (1.44 ×)
dash	48.4 s (1.12 ×)	118.4 s (1.10 ×)
bash	60.6 s (1.29 ×)	86.5 s (1.36 ×)
yash	62.2 s (1.30 ×)	93.7 s (1.41 ×)
zsh	722.4 s (1.36 ×)	911.9 s (1.62 ×)

```
: $((node = 0))
_get_val() { let node $2
  : $(($1 = _$((_heap + $2 + 1))))
  endlet $1 node
}
```

Binding positional parameters to variables.

```
_get_val() { # node: $2
  : $(($1 = _$((_heap + $2 + 1))))
}
```

Using positional parameters directly.

Fig. 5.4. Comparing a short function with and without the constant parameter optimization.

when compiled to shell. This overhead is particularly noticeable for functions that do minimal work, such as functions to allocate objects, as the `let` calls can end up taking longer than the rest of the function.

Fortunately, the POSIX specification includes local scoping for positional parameters – the initial value of a function parameter can be retrieved using its corresponding positional parameter. This feature can be used for parameters that remain constant throughout the execution of a function, eliminating the need to save them to the stack. This is done for the return variable, which is always `$1`. This allows functions without parameters and local variables to avoid the overhead of local variable management.

However, referring to parameters by their position can hurt readability, especially for large functions where the link between the parameter and its position may be difficult to follow. Since this optimization can bring a significant performance improvement, we compromise by letting the developer opt-in to this optimization by using the `const` type qualifier on the parameter. This makes it possible to use this optimization for small and frequently called functions, which we might be tempted to turn into macros in C (which would be even less readable). The mapping between positional parameters and C parameters is added as a comment to the function declaration to inform the reader of the correspondence, as shown in Figure 5.4. Since this optimization is used in `pnut-sh`, Table 5.2 shows the performance impact of *not* using this optimization.

Table 5.3. Execution times of `pnut-sh.sh` and `pnut-exe.sh` using `set` for local variables.

Shell	`pnut-sh.sh pnut-exe.c`	`pnut-exe.sh pnut-exe.c`
ksh	14.6 s (0.77 $\times$)	24.5 s (0.96 $\times$)
dash	37.5 s (0.87 $\times$)	105.6 s (0.98 $\times$)
bash	37.4 s (0.80 $\times$)	58.7 s (0.92 $\times$)
yash	36.8 s (0.77 $\times$)	62.2 s (0.94 $\times$)
zsh	294.0 s (0.55 $\times$)	378.7 s (0.67 $\times$)

In a function, the positional parameters can also be updated using the `set` command. This command replaces the positional parameters with the arguments it receives and can be used to store local values by adding them to the positional parameters. This feature makes it possible to save the value of local variables in the positional parameters instead of saving them to the stack with `let`, greatly speeding up the calling convention. However, doing so results in much less readable and idiomatic function prologues and epilogues, with the additional positional parameters that can be hard to keep track of. Figure 5.3 shows how the local variables can be implemented using `set` for the Fibonacci function and compares it to the `let` implementation.

Due to the readability issue, this optimization was not used in `pnut-sh`. Instead, the `const` parameter optimization was preferred as an opt-in alternative, reducing the cost of the calling convention for functions with constant parameters. The `const` parameter optimization has the added benefit of completely removing the calling convention cost of the parameter, eliminating it from the prologue and epilogue of functions. Still, the performance of the `set` implementation is significantly better than using `let` (See Table 5.3) on both versions of pnut, and is therefore kept as a compilation option for users for whom the loss of readability is worth the performance improvement.

5.2.3. Function Declaration

Combining the callee-save calling convention with using a result variable, the prologue of the generated shell functions is minimal, consisting of calls to `let` followed by the initialization of the local variables if necessary. The `let` calls for parameters are placed on the same line as the function declaration, matching the position where parameters are declared in C and hiding the fact that function arguments are passed as positional parameters.

The epilogue is also short, with only an assignment to the result variable and a call to `endlet` before returning. The Fibonacci function in Figure 5.3 shows the prologue and epilogue and compares it to the `set` version of the same function.

```c
int eval(ast node) {
  int lval = eval(get_left(node));
  int rval = eval(get_right(node));
  switch (get_op(node)) {
    case '+':
      return lval + rval;
    case '-':
      return lval - rval;
    default:
      return 0;
  }
}
```

C source code.

```sh
_eval() { let ast $2; let node $3
  let lval; let rval; let __t1
  _get_left __t1 $node
  _eval lval $__t1
  _get_right __t1 $node
  _eval rval $__t1
  _get_op __t1 $node
  case $__t1 in
    $__PLUS__)
      : $(($1 = lval + rval)) ;;
    $__MINUS__)
      : $(($1 = lval - rval)) ;;
    *)
      : $(($1 = 0)) ;;
  esac
  endlet $1 __t1 rval lval node ast
}
```

Tail position `return` optimization.

Fig. 5.5. Example of tail position `return` optimization.

5.2.3.1. Optimizations. To avoid repeating the function epilogue for every return statement, it can be useful to detect when a return statement is in tail position. In that case, the `endlet` parts of the epilogue can be omitted, since it already appears at the end of the function and is executed if the control is allowed to fall through to the end of the function. This optimization is useful for functions that branch and return values depending on the branch taken, reducing the noise of the calling convention and making the code more readable. This includes most functions that create and traverse the abstract syntax tree, which are very common in parsers and code generators. Figure 5.5 shows this optimization applied to a function evaluating constant expressions similar to the one used in `pnut-sh` to evaluate constant expressions.

Another common pattern is for the result of a function call to be immediately returned. In that case, the caller can pass its return variable to the callee so that the callee writes the result in the end location. This avoids the use of a costly local temporary variable in addition to saving an assignment from the temporary variable to the return location.

```
: $((_$((arr + i)) = 42)) # arr[i] = 42
: $((x = _$((arr + i))))  # x = arr[i]
```

Fig. 5.6. Contiguous arrays and random accesses in a POSIX shell.

5.3. Memory

Unlike `bash` and other more advanced shells, the POSIX standard doesn't provide arrays or any other data structure with indexing. The only way to store data is in variables, with each variable containing either a string or an integer (represented as a string). Fortunately, it is possible to define variables with a dynamic name using arithmetic expansion, which is how arrays and ultimately the heap can be implemented.

We assign each memory location the variable underscore followed by the address; address 0 corresponds to `_0`, address 1 corresponds to `_1`, etc. This prevents the use of C variables named with an underscore followed by a number, which are valid in C. We think this is an acceptable compromise, as the underscore is short and easily recognizable.

To reference dynamic variables, expansion is used to prefix the address with `_`. Arithmetic expansion can be nested multiple times and is expanded inside-out, permitting read and writes to dynamic variables, and so to memory locations. With this, pointers to objects are simply the address of the beginning of the object, just like in C. For example, writing and reading the array `arr` at index `i` is achieved like this:

5.3.1. Shell's Internal Representation of the Heap

The use of variables to represent the heap can cause performance issues. All shells (`dash`, `bash`, `ksh`, `zsh`, `mksh`, `yash`) store variables using one hash table for all variables, and so the time to access variables varies depending on the number of variables in the environment, which directly correlates with the amount of memory allocated by the C program. A consequence of this internal representation of the heap is that as more memory is allocated, the slower many of their operations get. Also, because memory location variables are pieced together dynamically, the shell must hash the variable name, which can impact access time as longer variable names take longer to hash. Consequently, smaller addresses are faster to access than larger ones.

5.3.2. Memory Management

Because shell variables don't need to be declared before they can be used, new memory locations can be created by simply accessing new variables. In turn, those memory locations

```
__ALLOC=1    # 0 is reserved for NULL
_malloc() { # $1 = return variable, $2 = size
  : $((_$__ALLOC = $2))        # Track object size
  : $(( $1 = __ALLOC + 1))     # Assign return value
  : $((__ALLOC += $2 + 1))     # Update bump pointer
}

_free() {                      # $2 = object to free
  __ptr=$(($2 - 1))            # Start of object
  __end=$((__ptr + _$__ptr)) # End of object
  while [ $__ptr -lt $__end ]; do
    unset "_$__ptr"
    : $((__ptr += 1))
  done
}
```

Fig. 5.7. `malloc` and `free` POSIX shell implementations.

are added to the shell environment when they are first written to. This separates the allocation of memory and the use of hash table slots.

This simplifies `malloc`'s implementation as it doesn't need to initialize memory and a bump allocator is sufficient to reserve addresses. This allows large memory objects to be allocated without impacting performance as their memory locations occupy the shell environment only if they are touched.

POSIX shell includes the `unset` command, which removes variables from the shell environment. This command is used to implement `free`, to reduce the size of the environment. We note that with the bump allocator design, `free` does not reduce the memory usage towards the memory limit, but because the POSIX standard requires signed long integer arithmetic, at least 2 GiB (2^{31}) can be accessed with this scheme. Figure 5.7 shows how `malloc` and `free` can be implemented.

5.4. Strings and I/O

Another difference between C and POSIX shell is how strings are represented. In C, strings are represented as null-terminated arrays of bytes, which can be used like any other array. In contrast, in shell, string values can be modified only using the provided interface. This interface is limited, being primarily useful for concatenating, cutting, and comparing strings, and it does not allow random access to characters – one of the most common access patterns used in C programs.

```
_puts() { # $2 = address of string to print
  __ptr=$2
  while [ $((__c = _$__ptr)) != 0 ]; do
    printf \\$((__c/64))$((__c/8%8))$((__c%8))
    : $((__ptr += 1))
  done
}

# $1 = address of string to pack
pack_string() { __str=$(_puts __ $1); }
```

Fig. 5.8. Packing a C string into a shell string.

5.4.1. String Representation

To reconcile this incompatibility, we settled on representing strings as arrays of integer character codes. This requires packing and unpacking the shell strings into arrays of bytes whenever a string crosses from C code to shell code and vice versa. This operation is required to initialize string literals or to do any I/O, which are very common for programs like compilers.

Converting from a C string to a shell string is relatively simple, as it can be done by outputting each character using the `printf \\`*octal_num* command and capturing the output in a command substitution as shown in Figure 5.8.

This conversion from C to shell string may be slow on certain shells as it uses a subshell. However, since almost all conversions are to then print the string, the subshell can be skipped and the characters printed directly.

Going the other way is more costly, as extracting the characters is difficult in POSIX shell. Using variable expansion, it is possible to remove the first character of a string using `tail=${string#?}` where "?" matches any character, and then remove this string from the original string using `${string%"$tail"}` to obtain the first character. This operation is repeated until the string is empty. Figure 5.9 shows how this can be done.

However, shells are likely not optimized for this specific use case, so we can expect that extracting 1 character requires traversing the whole string at least once, and potentially allocating a substring. This work is repeated for each character in the string, meaning that the time complexity of unpacking a string is $O(n^2)$, for n the length of the string. For particularly long lines, the quadratic time can cause considerable slowdown in the program's performance.

```
unpack_string() { # $1 = string to unpack
  __str=$1
  _malloc __addr $((${#__str} + 1))
  __ptr=$__addr
  while [ -n "$__str" ] ; do
    __tail=${__str#?}          # Remove head char
    __head=${__str%"$__tail"} # Get head char
    __byte=$(LC_CTYPE=C printf %d "'$__head")
    : $((_$__ptr = __byte))    # Store byte
    : $((__ptr += 1))          # Advance
    __str=$__tail
  done
  : $((_$__ptr = 0))
}
```

Fig. 5.9. Unpacking a shell string into a C string.

To mitigate this problem, we found that extracting longer chunks from long lines and iterating on those smaller substrings is much faster. Specifically, we break long lines into 256-character chunks, then again into 16-character chunks before extracting individual characters.

Extracting a character takes $O(n)$ operations only in the general case. Using POSIX regex patterns, it is possible to test whether a string is prefixed with a specific character. For shells that perform the relatively simple optimization that the asterix pattern (match any string) doesn't scan the rest of the string, matching on a specific prefix letter takes constant time. Fortunately, the ASCII character set is relatively small and an even smaller set of characters makes up the majority of C source code, and so hardcoding a list of the most common characters gives a significant speedup on most shells as it only resorts to the more expensive character extraction method for the less common characters. The function in Figure 5.10 features this optimization.

5.4.2. String Literals

Given the cost of unpacking strings, the initialization of string literals requires attention as initializing all strings at the beginning of the program could noticeably increase startup time. Also, each string initialization allocates memory which grows the environment and contributes to slowing down the execution. As a result, string literals are initialized the first time they are used, using the `defstr` function in Figure 5.11.

Each string literal is associated with a unique string variable that acts as a cache and is passed to `defstr`. When the variable is empty, the string is unpacked and the result is

```
# Unpack a Shell string into an appropriately sized buffer
unpack_string() { # $1: Shell string, $2: Buffer, $3: Ends with EOF?
  __fgetc_buf=$1
  __buffer=$2
  __ends_with_eof=$3
  while [ ! -z "$__fgetc_buf" ]; do
    case "$__fgetc_buf" in
      " "*) : $((_$__buffer = 32))  ;;
      "e"*) : $((_$__buffer = 101)) ;;
      "t"*) : $((_$__buffer = 116)) ;;
      # ...
      "*"*) : $((_$__buffer = 42))  ;;
      *)
        char_to_int "${__fgetc_buf%"${__fgetc_buf#?}"}"
        : $((_$__buffer = __c))
        ;;
    esac
    __fgetc_buf=${__fgetc_buf#?}      # Remove the first character
    : $((__buffer += 1))              # Move to the next buffer position
  done

  if [ $__ends_with_eof -eq 0 ]; then # Ends with newline and not EOF?
    : $((_$__buffer = 10))            # Line ends with newline
    : $((__buffer += 1))
  fi
  : $((_$__buffer = 0))               # Then \0
}
```

Fig. 5.10. Faster string unpacking in POSIX shell.

```
defstr() { # $1 = variable name, $2 = string
  if [ $(($1)) = 0 ]; then         # if variable is undefined
    unpack_escaped_string $1 "$2" # Unpack string and store in $1
  fi
}

defstr __str_0 "Hello"  # convert "Hello"
_puts __ __str_0        # print string
```

Fig. 5.11. C string literals in POSIX shell.

Table 5.4. Execution times of `pnut-sh.sh` and `pnut-exe.sh` using character codes directly.

Shell	`pnut-sh.sh pnut-exe.c`	`pnut-exe.sh pnut-exe.c`
ksh	18.3 s (0.97 $\times$)	25.3 s (0.99 $\times$)
dash	42.4 s (0.98 $\times$)	107.8 s (1.00 $\times$)
bash	45.8 s (0.98 $\times$)	62.4 s (0.98 $\times$)
yash	47.1 s (0.98 $\times$)	67.6 s (1.02 $\times$)
zsh	521.5 s (0.98 $\times$)	565.1 s (1.00 $\times$)

saved in the variable. The calls to `defstr` are placed just before the string literal is used, improving readability as the string variable is located close to where it is used.

5.4.3. Printf

C programs often output text with the `printf` function, which is usually implemented as a library function. Most calls to `printf` are with a constant format string, making it possible to decompose the function call into an equivalent sequence of calls to `_puts` and to the shell's `printf` function.

This makes it possible to exclude the `printf` implementation in the scripts produced by `pnut-sh`, and save many packing and unpacking of format strings and arguments. This optimization is particularly beneficial for programs that output a lot of text using `printf`, which includes `pnut-sh` itself.

5.5. Magic Numbers

We define magic numbers as numbers that appear in the shell code but not literally in the C code. These numbers can obfuscate the code and potentially hide bugs as they can be difficult to verify. These come from the use of character literals, enums and structures, and can be given meaning by assigning each to a `readonly` global variable with a descriptive name.

For character literals, the character codes are assigned to global variables named `__`*character*`__` for alphanumerical characters, and `__`*name*`__` otherwise. For example, `'A'` is given the `__A__` variable, and `' '` the `__SPACE__` variable. C character literals are then mapped to these shell variables to make the code more readable. This indirection adds at most a few percent to the execution time of the shell, as shown in Table 5.4, comparing with the character code placed directly in the code to the baseline.

5.5.1. Structures

Similarly, because structures are just like arrays, with the fields placed in consecutive memory locations which are accessed by adding the corresponding offset to the structure's base address, the offset of each field is computed by pnut and assigned to a global variable. For example, here is a C program with `structs` and the generated shell code:

```c
struct Point { int x; int y; };

struct Point *p;

void init_point() {
  p = malloc(sizeof(struct Point));
  p->x = 0;
  p->y = 0;
}
```

```sh
# Point struct member declarations
readonly __x=0
readonly __y=1
readonly __sizeof__Point=2

_p=0

init_point() {
  _malloc _p $__sizeof__Point
  : $((_$((_p + __x)) = 0))
  : $((_$((_p + __y)) = 0))
}
```

Fig. 5.12. Compiling C structs to POSIX shell.

5.6. Readability

The auditing of `pnut-sh.sh` being an obligatory step to ensure that the existing C compiler hasn't tampered in any way with the generated shell code, `pnut-sh` can include the C source code as comments in the shell code. With this option, each top-level declaration is prefixed with a comment containing the C code from which it was generated. This includes comments from the C code that often contain important information about the code, such as the purpose of a function or the meaning of a variable. This makes it easy to see the correspondence between the C code and the shell code and to verify that the shell code is correct.

Because it can almost double the size of the scripts generated, this option is turned off by default and is only used when generating `pnut-sh.sh` using an existing compiler to ensure that the generated shell code is correct.

Table 5.5. Time to initialize an array of size N.

N	`ksh`	`dash`	`bash`	`yash`	`zsh`
10000	0.03 s	0.03 s	0.05 s	0.06 s	0.05 s
20000	0.05 s	0.07 s	0.09 s	0.11 s	0.10 s
40000	0.10 s	0.24 s	0.19 s	0.20 s	0.18 s
80000	0.19 s	0.84 s	0.38 s	0.40 s	0.36 s
160000	0.40 s	2.92 s	0.71 s	0.81 s	0.71 s
320000	0.80 s	11.28 s	1.51 s	1.61 s	1.43 s

5.7. Performance Portability

In general, we observe that the fastest shell is `ksh`, with `bash` and `yash` in close second place, while the slowest shells are `dash` and `zsh`. This rule is not absolute, as the performance of a shell can vary greatly depending on the nature of the programs it runs.

5.7.1. Memory Use

As detailed earlier, performance can degrade when the shell environment grows too large, which is proportional to the amount of memory used by the program. This degradation highly depends on the shell, with `ksh` showing almost no sign of performance degradation, while `dash` quickly shows linear access times to variables. Table 5.5 shows the time to initialize an array of variable size.

To maximize portability, programs should limit their memory usage or delay it until necessary. A few ways to do this are to reuse memory locations, free memory as soon as possible, and leave memory uninitialized for as long as possible. In `pnut-sh`, the statically allocated arrays can be left uninitialized using a compile-time option to avoid increasing the environment size immediately on startup. This option is turned off by default, as it deviates from the C standard, where reading uninitialized memory produced an indeterminate value, but is used for `pnut-sh` and `pnut-exe` to speed them up significantly.

5.7.2. Faster String Conversion

The shell string to C string conversion plays a central part in all the input primitives of `pnut-sh` and can take a significant time for programs that read from the standard input or files, even when the lines are short. This is because opening a subshell to convert each character is slow, and on certain shells becomes even slower as the environment grows, as Table 5.6 shows.

Table 5.6. Time to convert characters using a subshell compared to a lookup table.

	ksh	dash	bash	yash	zsh
subshell (1000)	0.06 s	3.38 s	7.11 s	5.27 s	5.42 s
subshell (10000)	0.06 s	3.85 s	9.09 s	5.96 s	5.83 s
subshell (100000)	0.07 s	6.96 s	25.50 s	12.51 s	10.09 s
fast (1000)	0.01 s	0.01 s	0.04 s	0.15 s	0.03 s
fast (10000)	0.01 s	0.03 s	0.04 s	0.15 s	0.04 s
fast (100000)	0.02 s	0.02 s	0.03 s	0.15 s	0.05 s

Instead, a lookup table can be used to convert characters to their character code. This works for alphanumerical characters that can form valid variable identifiers, as the lookup can be done using variable expansion. For other characters, the case statement can be extended to match them. The subshell is then required only for control characters and extended ASCII characters, which are much rarer.

```
__c2i_0=48 ... __c2i_z=122

__c="A" # Convert character to character code
case $__c in
  [a-zA-Z0-9]) __code=$((__c2i_$__c)) ;;
  " ") __code=32 ;;
  *) __code=$(LC_CTYPE=C printf %d "'$__c") ;;
esac
```

Fig. 5.13. Character to code conversion using a lookup table.

This method (fast) is much faster on all shells, which can be seen in Table 5.6 that compares the original method (subshell) to the one using the lookup table when converting the characters of a string containing a mix of alphanumerical and special characters. The performance of both methods is influenced by the size of the environment, therefore, measures with 1000, 10000 and 100000 variables are shown.

Chapter 6

Evaluation

This chapter evaluates the usefulness of pnut and, more generally, of the POSIX shell as a platform for reproducible builds. It covers the auditability, portability, distribution and performance of `pnut-sh`, `pnut-exe`, `jammed.sh` and the pnut libc, and then compares our reproducible build approach to live-bootstrap's and Guix's full-source bootstrap.

For reference, performance is measured in the same environment as in Chapter 5, which corresponds to a typical development environment with recent versions of various shells (see Figure 5.1). The compilation options used when creating shell scripts are the same ones that are used for the baseline times of Chapter 5.

Pnut is designed to be a practical tool for compiling C to POSIX shell. It meets the goals of auditability, portability and reasonable performance, and has enough support of the C language for `pnut-sh` to compile itself and `pnut-exe` in a reasonable time, which demonstrates that POSIX shell is a viable "root language" for bootstrapping larger toolchains.

6.1. Auditability

By auditability, we refer to the ease with which a programmer can read and understand all the individual pieces of the reproducible build process and how they fit together. An auditable system is one that can be audited by a programmer without prior knowledge of the system in a reasonable amount of time. Auditability obviously includes code readability, but also the amount of code that is involved in the reproducible build process and its overall complexity.

6.1.1. Readability

The question of readability primarily concerns `pnut-sh.sh` as it is the only precompiled file in the reproducible build process. The script serves as the seed for the reproducible

Table 6.1. Size of `pnut-sh` and `pnut-exe` C source file, shell script, runtime and whitespace.

Pnut version	C source	Shell code	Runtime	Whitespace
`pnut-sh.c`	4263	6211 (1.46 ×)	429	236
`pnut-exe.c` (i386 Linux)	4089	5687 (1.39 ×)	431	284
`pnut-exe.c` (amd64 Linux)	4159	5791 (1.39 ×)	431	293
`pnut-exe.c` (amd64 macOS)	4199	5807 (1.38 ×)	431	293

build, so it is important that it can be reviewed and understood by a programmer. While readability is subjective, `pnut-sh` generates shell code that has the following properties:

- It follows the structure of the original C code, with indentation and using the same names for variables and functions.
- The code is regular, simple and predictable.
- It can include the C code as comments for auditing.

As a result, the `pnut-sh.sh` script can be read and audited by a programmer familiar with the C language without much difficulty (Figure 6.1 gives an example of a `pnut-sh.sh` function, annotated with the original C code as generated by `pnut-sh`). As a rough measure of readability, we compare the number of lines of code of the generated shell script (with no C source code annotations) to the size of the original C code in Table 6.1. For a fair comparison, the C source code was preprocessed and stripped of its comments and empty lines. The `pnut-sh.sh` and `pnut-exe.sh` shell scripts have between 38% and 46% more lines of code than the C code. This increase in size includes the runtime functions that are not present in the C code, as well as whitespace that separate declarations, making the actual difference smaller in practice.

Additionally, we went through the exercise of auditing `pnut-sh.sh`, comparing each compiled declaration to its C source code counterpart to ensure no trusting trust attacks took place. This process took 10 hours of work, at a rate of approximately 600 lines of code per hour, which is reasonable since it must only be done once. On subsequent uses of the script, the hash of the script can be verified (using `sha256sum.sh` bundled in `jammed.sh`) instead.

6.1.2. Code Complexity

Like readability, code complexity is subjective. However, the compiler was kept intentionally simple – a preprocessor supporting only the most common features, leaving out support for variadic and self-referential macros, a simple recursive descent parser that follows the C99 grammar definition directly with few exceptions, and code generators that perform little to no optimisation. Additionally, because the subset of C supported by `pnut-sh` is limited,

```
# int accum_digit(int base) {
#   int digit = 99;
#   if ('0' <= ch && ch <= '9') {
#     digit = ch - '0';
#   } else if ('A' <= ch && ch <= 'Z') {
#     digit = ch - 'A' + 10;
#   } else if ('a' <= ch && ch <= 'z') {
#     digit = ch - 'a' + 10;
#   }
#   if (digit >= base) {
#     return 0; /* char is not a digit in that base */
#   } else {
#     val = val * base - digit;
#     get_ch();
#     return 1;
#   }
# }
: $((digit = base = 0))
_accum_digit() { let base $2
  let digit
  digit=99
  if [ $__0__ -le $_ch ] && [ $_ch -le $__9__ ] ; then
    digit=$((_ch - __0__))
  elif [ $__A__ -le $_ch ] && [ $_ch -le $__Z__ ] ; then
    digit=$(((_ch - __A__) + 10))
  elif [ $__a__ -le $_ch ] && [ $_ch -le $__z__ ] ; then
    digit=$(((_ch - __a__) + 10))
  fi
  if [ $digit -ge $base ] ; then
    : $(($1 = 0))
  else
    _val=$(((_val * base) - digit))
    _get_ch __
    : $(($1 = 1))
  fi
  endlet $1 digit base
}
```

Fig. 6.1. C to POSIX shell compilation of the `accum_digit` function used in pnut.

the C language used in the implementation is straightforward, making the code easier to understand.

The same applies to the pnut libc, which implements the functions strictly required to bootstrap TCC. Its implementation favors simplicity over performance or completeness, and so is relatively short with only 1609 lines of code (see Table 6.2) and easy to read.

6.1.3. File Sizes

File size is another important aspect of auditability. Table 6.2 presents the size in lines of code and in kilobytes (KB) of the files used for the TCC bootstrap. The `jam.sh` archive is used in the preparation of the bootstrap seed (`jammed.sh`), `bootstrap.sh` is used to automate the bootstrap after the initial extraction of the files, `sha256sum.sh` is a shell implementation of the `sha256sum` utility used to verify the integrity of the extracted files, `bintools.c` contains the source code of basic file utilities used during the bootstrap, the `libc` directory contains the pnut libc source and header files, `tcc_patches` contains patches required to bootstrap TCC using pnut, and `tcc` are the C source files used for the i386 Linux bootstrap of TCC.

The largest file is the `jammed.sh` archive. It is larger than every other file combined, which is expected since every file is embedded as-is in the archive. Additional code and data, such as file names and paths, are also required to perform the extraction, resulting in a 2.5% size overhead. It is possible to recreate it from the `jam.sh` script, meaning its large size is not a concern.

Other large files are `pnut-sh.sh` and `pnut-exe.c`. A lot of the code is shared between them, as `pnut-sh` and `pnut-exe` share the same compiler frontend, with the only difference being that one is in C and the other in POSIX shell. Also, compared to TCC's source code, the pnut files contain many times fewer lines of code, putting pnut's size in perspective.

Table 6.2. Size of shell scripts and C source files used for the TCC bootstrap. The total row is the sum of all files in the `jammed.sh` archive, with some files being included more than once.

File	Description	Lines of code	Size (KB)
`jam.sh`	To create jam archive	270	6.9
`bootstrap.sh`	Bootstrap from shell script	229	9.7
`sha256sum.sh`	Shell implementation of sha256sum	301	9.6
`pnut-sh.sh`	Bootstrap seed	6211	194
`pnut-exe.c`	`pnut-exe` source files (i386 Linux)	9426	289
`bintools.c`	Basic file utilities	2038	68
`libc/includes`	pnut libc header files	391	8.1
`libc/src`	pnut libc source files	1218	25
`tcc_patches`	Patches for Tiny C Compiler	110	2.2
`tcc`	Tiny C Compiler i386 Linux source Files	28937	920
Total	Total	50250	1556
`jammed.sh`	Jam archive containing every other files	51186	1595

6.2. Portability

`Pnut-exe` can be bootstrapped from `pnut-sh.sh` on all tested shells, despite some shells deviating slightly from the POSIX standard. It has been tested on the shells `bash`, `dash`, `osh`, `ksh`, `yash` and `zsh`, and on Linux (x86_64), macOS (ARM), and Windows (x86_64 on WSL), all of which can run `pnut-sh.sh` and compile `pnut-exe` without any issues. This includes `bash` version 2.05b from 2002, which demonstrates the stability of the POSIX shell standard and the non-reliance on more modern shell features and bashisms. This is not surprising as no external utilities are used, nor any platform-specific code, making `pnut-sh.sh` exceptionally portable.

Once the executable version of `pnut-exe` is obtained, the shell is no longer required and portability is no longer defined by it, but by the targets supported by `pnut-exe`. `Pnut-exe` supports both 32-bit (i386) and 64-bit x86 (amd64), and the ELF and Mach-O file formats, which Linux and macOS use. The TCC bootstrap process has only been tested on i386 Linux; however, TCC supports many more targets, opening the door to a broader range of platforms.

6.3. Packaging and Distribution

The packaging and distribution of files required for a reproducible build is an important aspect of the reproducible build process, as it is only as secure as the bootstrap environment in which it executes. We solve this problem with `jam`, which packs all the files required to bootstrap TCC, the `pnut-sh.sh` seed script, `pnut-exe`'s source files, the pnut libc, `bintools.c` and TCC source files, into a single self-extracting archive (`jammed.sh`) that only requires a POSIX shell to extract. This archive may be safely unpacked without executing it using the `sift.sh` script, and its files audited with the `more.sh` pager, both of which can be typed manually because of their small size (less than 30 lines). With these tools, more complete tools such as `sha256sum.sh` can then be extracted and used to verify the integrity of the archive before it is executed.

Because pnut already depends on the shell, this packaging method does not introduce any additional risk compared to traditional distribution methods, which use tools such as `tar`. Additionally, having the process start from a single self-extracting file facilitates its distribution, as the user only needs to securely transfer a single file to their system to reproduce the build environment. Outside our reproducible build processes, shell scripts produced by `pnut-sh`, including `pnut-sh.sh`, are also single files that can be easily distributed and installed.

6.4. Performance

The majority of the time spent in the reproducible build process is in the execution of the shell scripts, which is expected as the shell is much slower than native executables. In our reproducible build, the executed shell scripts are `jammed.sh` to first extract the files, `pnut-sh.sh` to compile `pnut-exe.c` and finally `pnut-exe.sh` to recompile `pnut-exe` and get a statically linked executable that is no longer slowed down by the shell. Because `jammed.sh` is not a compiler, it is evaluated separately from `pnut-sh.sh` and `pnut-exe.sh`.

6.4.1. Pnut Compilation Performance

Pnut's performance can be evaluated in two ways: the time taken by `pnut-sh.sh` and `pnut-exe.sh` to compile a C program to shell and the execution time of the generated shell script.

In addition to the time taken by `pnut-sh.sh` and `pnut-exe.sh` to compile `pnut-exe.c`, we also measure the time to compile smaller programs using `pnut-sh.sh`. Because some of the programs require features outside of the subset supported by the minimal `pnut-sh.sh`, we use the fully-featured pnut variants for these tests. These programs include a simple hello world (`hello.c`), a program that prints the first 20 Fibonacci numbers (`fib.c`), a file copy program (`cp.c`), a file reading program (`cat.c`), a program counting the number of characters, words and lines of a file (`wc.c`), a program computing the SHA256 of a file (`sha256sum.c`), the C4 compiler [41] [1] (`c4.c`) and the Ribbit Virtual Machine [37] (`repl.c`) running a R4RS Scheme REPL. The compilation times are shown in Table 6.3, which also includes, for reference, the time taken to compile the same programs by the `pnut-sh` executables produced by GCC (with the `-O3` optimization level) and by `pnut-exe` (i386 version).

Except for small programs, the execution of `pnut-sh.sh` takes a few seconds even on the fastest shells. These long compilation times mean that using `pnut-sh.sh` directly is not practical for compiling large programs. The slow compilation is due to the shell's slow execution speed, as `pnut-sh` compiled with GCC and `pnut-exe` are much faster and can compile large programs in a reasonable amount of time. `pnut-sh.sh` being slow is not an issue since it is only meant to compile the faster `pnut-exe` executable.

The execution times of the generated shell files are shown in Table 6.4. Again, we include for reference executables produced by GCC (with the `-O3` optimization level) and from `pnut-exe`. C4 is given its own source code as input, the R4RS REPL receives a simple hello world program and other programs that take a file as input use a 64KiB file with 512 characters per line and 128 lines. The execution time of the `hello.c` is almost instant,

[1]The `c4.c` compiler is a minimal C compiler that can compile itself. It can be compiled by pnut with the addition of a single memset call and adjusting some buffer sizes.

Table 6.3. Compilation times of programs using `pnut-sh.sh` per shell. The times taken for the `pnut-sh` executable made from `GCC` and `pnut-exe` to compile the same programs are included for comparison.

	`ksh`	`dash`	`bash`	`yash`	`zsh`	`pnut-sh` (GCC)	`pnut-sh` (`pnut-exe`)
`hello.c` (11 LOC)	< 0.05 s	< 0.05 s	0.1 s	0.1 s	0.1 s	0.001 s	0.001 s
`fib.c` (48 LOC)	0.1 s	0.1 s	0.2 s	0.2 s	0.4 s	0.001 s	0.003 s
`cp.c` (47 LOC)	0.1 s	0.1 s	0.3 s	0.3 s	0.6 s	0.001 s	0.017 s
`cat.c` (48 LOC)	0.1 s	0.1 s	0.4 s	0.3 s	0.7 s	0.001 s	0.018 s
`wc.c` (74 LOC)	0.2 s	0.2 s	0.5 s	0.5 s	1.0 s	0.001 s	0.019 s
`sha256sum.c` (244 LOC)	1.1 s	1.0 s	2.9 s	2.8 s	6.8 s	0.001 s	0.029 s
`c4.c` (539 LOC)	3.6 s	5.0 s	10.3 s	9.8 s	44.7 s	0.003 s	0.087 s
`repl.c` (894 LOC)	4.0 s	9.2 s	11.8 s	11.6 s	71.9 s	0.003 s	0.085 s
`pnut-sh.c` (9076 LOC)	25.1 s	73.4 s	64.0 s	63.4 s	1060.9 s	0.007 s	0.560 s
`pnut-exe.c` (9426 LOC)	23.1 s	58.6 s	59.0 s	58.5 s	761.2 s	0.007 s	0.486 s

Table 6.4. Execution times of small programs produced by `pnut-sh.sh` per shell. The time taken for the same executables compiled by `GCC` and `pnut-exe` are included for comparison.

	`ksh`	`dash`	`bash`	`yash`	`zsh`	`GCC`	`pnut-exe`
`hello.c` (11 LOC)	< 0.05 s	< 0.05	< 0.05	< 0.05	< 0.05	0.001 s	0.001 s
`fib.c` (48 LOC)	0.7 s	0.4 s	1.7 s	1.9 s	2.6 s	0.001 s	0.001 s
`cp.c` (47 LOC)	2.2 s	1.0 s	3.2 s	3.9 s	4.2 s	0.002 s	0.001 s
`cat.c` (48 LOC)	2.2 s	1.0 s	3.2 s	4.0 s	4.1 s	0.001 s	0.001 s
`wc.c` (74 LOC)	2.7 s	1.4 s	4.5 s	5.6 s	7.3 s	0.002 s	0.002 s
`sha256sum.c` (244 LOC)	5.6 s	2.2 s	7.7 s	8.5 s	6.7 s	0.003 s	0.008 s
`c4.c` (539 LOC)	1.1 s	2.1 s	3.0 s	3.0 s	3.6 s	0.002 s	0.002 s
`repl.c` (894 LOC)	1.9 s	14.8 s	3.5 s	3.6 s	14.1 s	0.002 s	0.003 s

indicating that the initialization time of the generated script is minimal, `fib.c` takes a few seconds, which is expected as it does mostly arithmetic and function calls, two areas where the shell is not particularly good. The other scripts that perform I/O are between 1000 and 10000 times slower than the executables produced by GCC, but still only take a few seconds to read and output a file or to count the number of characters, words and lines of a file. The `sha256sum` and `repl` programs are the slowest as they combine I/O with many arithmetic operations, but are still usable on small inputs.

6.4.2. `Jam` Performance

We measure the time taken by `jammed.sh` to extract the files (listed in Table 6.2) required for bootstrapping TCC. In total, these files use 1556 KB of disk space, and, when packaged into `jammed.sh` as text files, occupy 1595 KB. We also test the time taken when the same files are base64-encoded, evaluating the cost of extracting binary files compared to text. The `jammed.sh` archive with base64-encoded files is 2141 KB in size, representing a 37.6% size increase primarily due to the base64 encoding overhead. This test is pessimistic, as the majority of files in jam archives are text files, and the binary files, if any, will likely be archives providing some compression, which compensates for the increased cost of binary files. The results are shown in Table 6.5.

Unlike during the reproducible build process, the extraction is done all at once, using a precompiled `bintools`, since we only want to measure the time taken by the extraction and not the rest of the bootstrap process.

Table 6.5. Time to extract the files required for bootstrapping TCC from `jammed.sh`. Both encoding methods are shown: text files being encoded as-is using here-documents, and base64-encoded files being decoded during extraction.

Shell	Text files	Base64-encoded file
ksh	0.3 s	49.9 s
dash	0.9 s	13.8 s
bash	0.6 s	56.5 s
yash	0.7 s	81.4 s
zsh	1.2 s	58.5 s

We can see that the time taken by `jammed.sh` to extract the textual files is reasonably fast, taking around a second on all shells. This is well below the time it takes to bootstrap `pnut-exe` from `pnut-sh.sh`, showing that shell archives are a viable option for packaging and distributing bootstrapping files.

For binary files, the situation is less favorable, with extraction times significantly higher across all shells and approaching the bootstrap time of `pnut-exe` on certain shells. This makes jam archives less practical for binary files, but still usable considering the benchmark evaluates the worst-case scenario. In practice, this will likely not be an issue since jam archives are relevant in contexts where tools such as `tar` and `sha256sum` are not yet available. Once they are bootstrapped, large binary archives may be sourced and unpacked securely using the bootstrapped `bintools`, without the performance overhead of jam archives.

6.5. Use for Reproducible Builds

As demonstrated in Chapter 3, TCC can be bootstrapped from only a POSIX shell and human-readable source files. Additionally, we have reproduced the TCC executable produced by live-bootstrap, confirming that our reproducible build process is correct and may be used to bootstrap a fully functioning Linux environment.

Live-bootstrap and Guix's full-source bootstrap are very similar, both using the `stage0` project as the starting point and `Mes` to compile TCC, and therefore are more or less the same for our comparison. We'll refer to both of them as *live-bootstrap* in the rest of this section. We compare pnut and these two approaches along the dimensions of auditability, portability and performance.

6.5.1. Auditability

As defined in Section 6.1, auditability is the ease with which a programmer can read and understand all the individual pieces of the reproducible build process and how they fit together. As a result, we must consider the size of the seeds, of the source files and all auxiliary scripts used in a reproducible build.

For pnut, the seed script is `pnut-sh.sh`, which is 6211 lines of code, with around 22000 lines of code when counting every other file described in Table 6.2, excluding TCC's source files.

For live-bootstrap, the seed script is the `hex0` executable, which stands at 190 bytes at the time of writing. The rest of the system is bootstrapped from the following source files (for i386 POSIX platforms):

- The `kaem-minimal` shell, implemented in the hex0 language in 422 lines used to automate the bootstrapping process.
- `hex1` and `hex2`: basic assemblers *without* mnemonics that can compute relative and absolute offsets, written in 317 lines of hex0 and 620 lines of hex1 code, respectively.
- M0: an assembler that allows macros to be defined and used instead of hex codes, written in 858 lines of hex2.
- `cc_x86`: a C compiler written in 4400 lines of M0 code.
- M2-Planet: The PLAtform NEutral Transpiler implemented in 8500 lines of C code.
- Mescc-tools: A collection of tools used to be used with Mes, written in around 2500 lines of C code.
- Mes: A Scheme interpreter implemented in 5000 lines of C code.
- MesCC: A C compiler implemented around 8000 lines of Scheme code.
- The Mes C library, around 16000 lines of C code.

The total size of the source files in live-bootstrap is about 45000 lines of code, twice the size of pnut. The number of components is also larger, with many tools being necessary to reach a language that resembles C, written in more than 5000 lines of assembly-like code. Auditing the source code of these tools requires a strong understanding of the instruction set architecture and the operating system they are designed to run on. Additionally, the low-level nature of these languages also makes reviewing them more difficult, as few guardrails are in place to prevent bugs or unintended behavior from slipping past the reviewer's eyes. Fortunately, after `cc_x86` is compiled, a subset of the C language can be used, and the subsequent source code can be audited with the same difficulty as pnut's source code.

6.5.2. Portability

For a reproducible build to be portable, every step along the way must either be platform-independent or have support for the targeted platform. For pnut, the only platform-specific step when bootstrapping TCC is the `pnut-exe` executable, the few primitive functions of the libc written in inline assembly and TCC (otherwise it wouldn't be able to recompile itself), as shown in Figure 6.2. The `pnut-sh.sh` script is inherently portable as it is a POSIX shell script and can be run on any POSIX-compliant shell. This simplifies the process of supporting additional platforms, as only `pnut-exe` needs to be extended while `pnut-sh.sh` stays the same. The portability of `pnut-sh.sh` also helps the auditability of the process, as only one seed script is used for all platforms. In contrast, the live-bootstrap approach requires a different seed and source files for every platform, which means that supporting a new platform requires using a distinct stage0 bootstrap that needs to be audited.

Fortunately, instruction set architectures and operating systems are stable and limited in number, which limits the number of platforms to support and makes this problem manageable. Additionally, supporting new platforms requires extending all the platform-specific tools, such as compilers, further down the bootstrap chain, involving the work of many people from many different projects.

6.5.3. Time to Bootstrap TCC

For our reproducible build to be practical, it must be sufficiently fast not to disproportionally slow down the build process of the software that uses it. Performance is not as important as the other factors, as bootstrapping the toolchain is only required the first time it is built, with subsequent builds able to reuse part of the bootstrapped toolchain.

Once `pnut-exe` is obtained, compiling TCC takes only a few seconds, which is fast enough to be practical. As a result, the time taken to bootstrap TCC from `pnut-sh.sh` is dominated

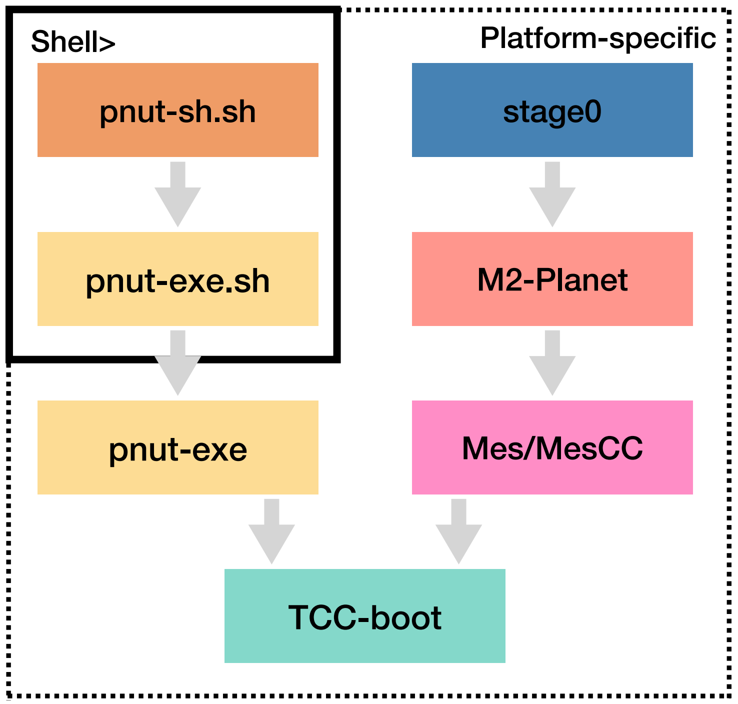


Fig. 6.2. Platform-specific components of pnut and of reproducible builds based on `stage0`.

Table 6.6. Time to bootstrap TCC using pnut on each shell, split into the four steps: the `jammed.sh` extraction, the compilation of `pnut-exe` from `pnut-sh.sh`, the recompilation of `pnut-exe` from `pnut-exe.sh`, and the TCC bootstrap.

Shell	`jammed.sh`	`pnut-exe.sh`	`pnut-exe`	TCC	Total
ksh	0.3 s	18.4 s	25.4 s		44.2 s
dash	0.9 s	44.1 s	112.3 s		156.7 s
bash	0.6 s	46.4 s	63.3 s	~5 s	110.1 s
yash	0.7 s	47.8 s	67.6 s		97.8 s
zsh	1.2 s	522.9 s	556.2 s		1009.7 s

by the time taken to bootstrap `pnut-exe`. Table 6.6 shows the time taken for each step, from the extraction of the files to the final TCC executable for each shell.

For live-bootstrap, bootstrapping stage0 is more or less instant, and the time taken to bootstrap TCC is dominated by the time taken to compile TCC using the MesCC compiler, which runs on the Mes Scheme AST interpreter. In total, this process takes about 10 minutes, a comparable amount of time to the time taken by pnut to bootstrap TCC on all shells except `zsh`.

6.6. Complementarity with Other Reproducible Builds

While the comparison with the live-bootstrap approach is worthwhile, it is important to keep in mind that both approaches are complementary and serve different purposes. The live-bootstrap approach is designed as a self-contained bootstrap process for a complete system, while our reproducible build process is focused on providing a practical way to bootstrap a C toolchain. Different users with different needs will find one or the other more suitable to their situation, or may even use both to increase the robustness of their reproducible build process, effectively doing diverse double-compilation between the two bootstrapping paths.

Pnut and other reproducible build projects also contribute to each other, as bootstrapping paths can be convoluted and difficult to find and establish. Pnut has greatly benefited from the work done by others, and counts on the existing bootstrapping path between TCC and the rest of the C toolchain.

In the other direction, pnut may also contribute to other projects, specifically to live-bootstrap, by providing an alternative to Mes and MesCC and potentially speeding up the bootstrap process. Pnut may also be used to solve the initial environment problem for other projects that use pre-built binaries to prepare their initial environment. Taking Guix as an example, as noted by Tournier [43], even if its full-source bootstrap process is rooted in the 190 bytes hex0 seed from stage0, Guix itself requires the binaries for the `Guile` Scheme compiler, `bash`, `tar`, `mkdir` and `xz`, totaling 25 MB in size, to initiate the bootstrap process. Using pnut, these binaries could be bootstrapped from `bash`, removing the need to distribute them as pre-built binaries and reducing the set of seeds to only the shell and `hex0`.

Chapter 7

Future Work

This chapter discusses potential improvements and extensions to the work presented in this thesis, and how they could make pnut a more practical tool for reproducible builds.

7.1. Smaller Seed Script

The `pnut-sh.sh` script comes at 6211 lines of code. Reviewing this much code may discourage many, even if the code is well-organized and documented. Fortunately, `pnut-sh.c`, even in its minimal version, still contains many features only used by `pnut-exe.c` for compiling TCC. With some effort, it should be possible to reduce the size of `pnut-sh.sh` by at least a few hundred lines of code, and more with additional bootstrapping steps. The resulting shell script would be smaller and easier to review, making it more approachable to interested developers.

7.2. Faster `pnut-exe` Bootstrap

The time required to bootstrap `pnut-exe` was a concern during the development of pnut and still is, as it takes from a few minutes to an hour on modern machines, depending on the shell implementation. We can imagine the same process on older machines taking much longer, possibly to the point of being impractical. The long bootstrap time is somewhat mitigated by the fact that it only needs to be done once, but a faster reproducible build process increases the chances it will be used in the first place.

Memory usage is the primary determining factor for the execution time of shell scripts, especially for the slower shells. As such, further optimizing the memory usage of both `pnut-sh.sh` and `pnut-exe.sh` could lead to lower bootstrap times. Since the code generators are one-pass and already optimized for memory efficiency, the pnut frontend is responsible for most of the memory usage, particularly the symbol table and abstract syntax tree (AST) nodes. Applying the same memory management strategy used by the code generators is likely the simplest and most effective option, with the objects allocated during the compilation of each top-level declaration being released using an arena-style allocator. Alternatively, strongly coupling the parser and code generators is another option that would avoid the use of intermediate data structures, at the cost of a more complex code base.

Zooming out from the implementation details of the compilers, the C language offers very little in the way of modularity and encapsulation, which can pose barriers to optimizations. In particular, the representation of strings in C as null-terminated arrays of characters, which allows pointers from anywhere in the program to access and mutate the string, is a source of inefficiency since it prevents the use of native strings in the target language.

Rewriting parts of pnut to use a string interface that hides the underlying representation may provide a solution. That way, `pnut-sh` could use the native shell string type and primitives to produce faster shell scripts. This would come at the cost of less idiomatic C code, however, which may make the code base harder to understand, as well as requiring a C implementation of the custom string type for compatibility with other C compilers. Pushing the idea further, exploring other programming languages such as Scheme for the implementation of pnut could also be an option. In addition to speeding up the execution of the compiled programs, using languages better suited for writing compilers could also lead to a smaller `pnut-sh.sh` seed script.

7.3. Support for More Platforms

One of the advantages of using POSIX shell as the root of our reproducible build process is its portability. However, to benefit from this portability and have reproducible builds for a wide range of platforms, `pnut-exe` needs to be ported to different architectures and operating systems. `Pnut-exe` should aim to support the CPU architectures and operating systems supported by TCC, as stepping outside this set would involve finding an alternative compiler to reach GCC from `pnut-exe`, a significant undertaking. As such, support for the ARM and RISC-V architectures on Unix-like operating systems should be possible with moderate effort.

Supporting more platforms also means more diversity of shell implementations to perform diverse double-compilation, strengthening our reproducible builds against trusting trust attacks.

Chapter 8

Conclusion

We have presented `pnut-sh`, a C to POSIX shell transpiler written in C that can compile itself and that generates human-readable shell code. Because the compiler is self-applicable, `pnut-sh` can be distributed as a human-readable shell script, `pnut-sh.sh`, that can run on any system with a POSIX-compliant shell. `pnut-sh.sh` can then be used to compile `pnut-exe`, a C to x86 compiler, that in turn can be used to produce a bootstrap version of the Tiny C Compiler (TCC).

We then discussed how a fully functional version of TCC can be bootstrapped from this bootstrap version of TCC, with a focus on bootstrapping floating-point numbers and large integers. This final version of TCC can then be used to bootstrap a complete build toolchain and development environment, including compilers, linkers, and other tools.

The initial environment problem was also outlined, with shell archives being proposed as a solution to this problem without introducing additional dependencies or risks. We presented the `jam` tool, our implementation of this solution, which can be used to create self-extracting shell archives using solely POSIX shell features, as well as tools to audit this packaging format.

Together, `pnut-sh`, `pnut-exe` and `jam` demonstrate that POSIX shell is a practical tool for reproducible builds. Additionally, the numerous, diverse and widely available POSIX shell implementations provide the ideal platform for diverse double-compilation, countering "trusting trust" attacks that traditional reproducible build processes are vulnerable to.

Finally, because POSIX shell is not a typical compilation target, we discussed the generation of human-readable POSIX shell code from C, preserving the structure and semantics of the original C code. Special attention was given to the performance of `pnut-sh.sh`, `pnut-exe.sh` and other generated scripts to ensure they are sufficiently fast on all shells. Performance pitfalls to avoid when using the shell as a compilation target are documented, along with methods to prevent them.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 1986. Visited on 2024-07-01.

[2] Kenneth Almquist. Almquist shell, 1989. URL `https://www.in-ulm.de/~mascheck/various/ash/`. Visited on 2024-07-01.

[3] Andrew W. Appel. Axiomatic bootstrapping: a guide for compiler hackers. *Transactions on Programming Languages and Systems*, 16(6):1699–1718, 1994. doi: 10.1145/197320.197336.

[4] Henry G. Baker. Shallow binding in lisp 1.5. *Commun. ACM*, 21(7):565–569, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359566. URL `https://doi.org/10.1145/359545.359566`.

[5] Morris Bolsky and David Korn. Kornshell, 1983. URL `https://github.com/ksh93/ksh`. Visited on 2024-07-01.

[6] Andy Chu. Oil shell, 2016. URL `https://github.com/oils-for-unix/oils`. Visited on 2024-07-01.

[7] Roland Clobus. Irregular status update about reproducible debian live iso images, 3 2025. URL `https://lists.reproducible-builds.org/pipermail/rb-general/2025-March/003675.html`. Visited on 2025-08-21.

[8] Bootstrappable Builds Contributors. Project - maintaining gcc version 4.7. URL `https://bootstrappable.org/projects.html`. Visited on 2025-08-21.

[9] Ludovic Courtès. Functional package management with guix, 2013. URL `https://arxiv.org/abs/1305.4584`.

[10] Ludovic Courtès and Janneke Nieuwenhuizen. The full-source bootstrap: Building from source all the way down, 4 2023. URL `https://guix.gnu.org/en/blog/2023/the-full-source-bootstrap-building-from-source-all-the-way-down/`. Visited on 2025-08-21.

[11] Devrandom. https://gitian.org, 1 2011. URL `https://gitian.org/`. Visited on 2025-08-21.

[12] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, page 79–92, USA, 2004. USENIX Association.

[13] Ayer et al. Stripnondeterminism is a perl library for stripping non-deterministic information such as timestamps and filesystem ordering from various file and archive formats, 2014. URL `https://salsa.debian.org/reproducible-builds/strip-nondeterminism`. Visited on 2025-08-21.

[14] Bellard et al. Tiny C compiler, 2001. URL `https://bellard.org/tcc/`. Visited on 2024-04-15.

[15] Bobbio et al. In-depth comparison of files, archives, and directories, 2014. URL `https://diffoscope.org/`. Visited on 2025-08-21.

[16] Falstad et al. Z shell, 1990. URL `https://www.zsh.org/`. Visited on 2024-07-01.

[17] Fox et al. Bourne-again shell, 1989. URL `https://git.savannah.gnu.org/cgit/bash.git`. Visited on 2024-07-01.

[18] Huberdeau et al. A self-compiling c transpiler targeting human-readable posix shell, 2024. URL `https://github.com/udem-dlteam/pnut`.

[19] Jędrzejewski-Szmek et al. Changes/package builds are expected to be reproducible, 03 2025. URL `https://fedoraproject.org/wiki/Changes/Package_builds_are_expected_to_be_reproducible`. Visited on 2025-08-21.

[20] Masters et al. Builder-hex0 is a minimal build system which includes a bootloader, kernel, shell, and a hex0 compiler, 2022. URL `https://github.com/ironmeld/builder-hex0`. Visited on 2025-08-21.

[21] Nieuwenhuizen et al. GNU Mes, 2016. URL `https://www.gnu.org/software/mes`. Visited on 2024-04-15.

[22] Nieuwenhuizen et al. Bootstrappable tcc/tinycc – tiny c compiler's bootstrappable fork, 2017. URL `https://gitlab.com/janneke/tinycc`. Visited on 2025-08-21.

[23] Orians et al. A set of minimal dependency bootstrap binaries, 2016. URL `https://github.com/oriansj/stage0`. Visited on 2025-08-21.

[24] Tyler et al. Use of a linux initramfs to fully automate the bootstrapping process, 2020. URL `https://github.com/fosslinux/live-bootstrap`. Visited on 2025-08-21.

[25] Watanabe et al. Yet another shell, 2009. URL `https://github.com/magicant/yash`. Visited on 2024-07-01.

[26] Xu et al. Debian almquist shell (dash), 1997. URL `https://git.kernel.org/pub/scm/utils/dash/dash.git`. Visited on 2024-07-01.

[27] Edmund Grimley Evans. Bootstrapping a simple compiler from nothing, 2002. URL `https://web.archive.org/web/20100303235322/http://homepage.ntlworld.`

com/edmund.grimley-evans/bcompiler.html. Visited on 2025-08-21.

[28] International Organization for Standardization. Iso/iec 9899:1999 - programming languages - c. Standard, International Organization for Standardization, 1999. URL https://www.iso.org/standard/29237.html. Visited on 2025-08-21.

[29] Michael Forney. cproc: Small c11 compiler based on qbe, 2019. URL https://sr.ht/~mcf/cproc/. Visited on 2025-08-21.

[30] John Gilmore. A history of reproducible builds in and around debian. URL https://reproducible-builds.org/docs/history/. Visited on 2025-08-21.

[31] John Gilmore. Source_prefix_map and occam's razor, 1 1990. URL https://lists.reproducible-builds.org/pipermail/rb-general/2017-January/000309.html. Visited on 2025-08-21.

[32] Dan Halbert. The early history of the more command, 1994. URL https://danhalbert.org/more.html.

[33] Laurent Huberdeau and Cassandre Hamel. Pnut: A c to posix shell compiler you can trust, 2024. URL https://pnut.sh.

[34] Laurent Huberdeau, Cassandre Hamel, Stefan Monnier, and Marc Feeley. The design of a self-compiling c transpiler targeting posix shell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*, SLE '24, page 70–83, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400711800. doi: 10.1145/3687997.3695639. URL https://doi.org/10.1145/3687997.3695639.

[35] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[36] Chris Lamb. Reproducible builds in august 2022, 8 2022. URL https://reproducible-builds.org/reports/2022-08/. Visited on 2025-08-21.

[37] Léonard Oest O'Leary, Mathis Laroche, and Marc Feeley. A r4rs compliant repl in 7 kb, 2023. URL https://arxiv.org/abs/2310.13589.

[38] *The Open Group Base Specifications Issue 7*. The Open Group, 2018. URL https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html.

[39] GNU Project. Gnu 'shar' utilities, 1994. URL https://www.gnu.org/software/sharutils/manual/sharutils.html#Introduction. Visited on 2025-08-21.

[40] Richard Stallman. What is free software?, 1986. URL https://www.gnu.org/philosophy/free-sw.html.en.

[41] Robert Swierczek. c4 - c in four functions, 2014. URL https://github.com/rswier/c4/tree/master. Visited on 2024-07-01.

[42] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8): 761–763, aug 1984. doi: 10.1145/358198.358210.

[43] Simon Tournier. Is guix full-source bootstrap a lie?, 10 2023. URL `https://simon.tournier.info/posts/2023-10-01-bootstrapping.html`. Visited on 2024-07-01.

[44] Rui Ueyama. chibicc: A small c compiler, 2019. URL `https://github.com/rui314/chibicc`. Visited on 2025-08-21.

[45] David Wheeler. Countering trusting trust through diverse double-compiling. In *Annual Computer Security Applications Conference*, page 33–48, 2005. doi: 10.1109/CSAC.2005.17.

Appendix A

Pnut README

In this appendix, we include part of the pnut README file, which provides installation and usage instructions for pnut. It is available in the pnut GitHub repository [18].

```
# Pnut: A Self-Compiling C Transpiler Targeting Human-Readable POSIX Shell

Pnut compiles a reasonably large subset of C99 to human-readable POSIX shell
scripts. It can be used to generate portable shell scripts writing shell code.

Try the web version at https://pnut.sh.

Its main uses are:

- As a transpiler to write portable shell scripts in C.
- As a way to bootstrap a compiler written in C with an executable version that
  is still human readable, for reproducible builds.

Main features:

- No new language to learn -- C code in, shell code out.
- The human-readable shell script is easy to read and understand.
- A runtime library including file I/O and dynamic memory allocations.
- A preprocessor (`#include`, `#ifdef`, `#if`, `#define`).
- Integrates easily with existing shell scripts.

## Install

Pnut can be distributed as the `pnut-sh.sh` shell script, or compiled to
executable code using a C compiler. Pregenerated shell scripts can be found on
the GitHub releases page.

To compile and install pnut:
```

```
> git clone https://github.com/udem-dlteam/pnut.git
> cd pnut
> sudo make install DESTDIR=/usr/local

This installs both ‘pnut-sh.sh‘ and ‘pnut‘ to ‘/usr/local/bin/‘.

Pnut also supports a native code backend that generates executable code (x86
Linux and MacOS for now), which we call ‘pnut-exe‘. To install ‘pnut-exe‘ and
its shell version ‘pnut-exe.sh‘:

> sudo make install-pnut-exe DESTDIR=/usr/local

### Compilation Options

Compilation options can be used to change the generated shell script:

- SH_ANNOTATE=1: include the original C code in the generated shell script.
- SH_COMPACT_RT=1: use compact runtime library (with reduced I/O performance).
- SH_FAST=1: generate shell code using ‘set‘ for local variables.
- MINIMAL=1: include only the set of features required to bootstrap pnut.

They can be set using ‘make install SH_ANNOTATE=1 ...‘.

## How to Use

The ‘pnut‘ compiler takes a C file path as input, and outputs to stdout the
POSIX shell code.

Here's an example of how to compile a C file using Pnut:

> pnut-sh.sh fib.c > fib.sh # Compile fib.c to a shell script
> chmod +x fib.sh           # Make the shell script executable
> ./fib.sh                  # Run the shell script
```

Listing A.1. Excerpt from the pnut README file.

Appendix B

`Hex0` x86 Assembly Source Code

In this appendix, we include the complete x86 assembly source code of `hex0`. It is a minimal x86 ELF executable that reads hexadecimal nibbles from a file and writes their binary representation to an output file. It is the binary seed used by the `stage0` [23] project to bootstrap a minimal C compiler from which more complete compilers can be built. Each line corresponds to a machine instruction, in hexadecimal form, followed by its mnemonic representation and a comment describing its purpose.

```
# SPDX-FileCopyrightText: 2019 Jeremiah Orians
# SPDX-FileCopyrightText: 2022 Andrius Stikonas
# SPDX-FileCopyrightText: 2024 Noah Goldstein
#
# SPDX-License-Identifier: GPL-3.0-or-later

## ELF Header
#:ELF_base
7F 45 4C 46                 # e_ident[EI_MAG0-3] ELF's magic number

01                          # e_ident[EI_CLASS] Indicating 32 bit
01                          # e_ident[EI_DATA] Indicating little endianness
01                          # e_ident[EI_VERSION] Indicating original elf

03                          # e_ident[EI_OSABI] Set at 3 because FreeBSD is strict
00                          # e_ident[EI_ABIVERSION] Set at 0 because no one cares

00 00 00 00 00 00 00        # e_ident[EI_PAD]

02 00                       # e_type Indicating Executable
03 00                       # e_machine Indicating x86
01 00 00 00                 # e_version Indicating original elf

4C 80 04 08                 # e_entry Address of the entry point
2C 00 00 00                 # e_phoff Address of program header table
00 00 00 00                 # e_shoff Address of section header table

00 00 00 00                 # e_flags

34 00                       # e_ehsize Indicating our 52 Byte header

20 00                       # e_phentsize size of a program header table

# The following 8 bytes are shared by both ELF header and program header.
```

```
## Program Header
#:ELF_program_headers
#:ELF_program_header__text
01 00                       # e_phnum number of entries in program table

00 00                       # e_shentsize size of a section header table
00 00                       # e_shnum number of entries in section table

00 00                       # e_shstrndx index of the section names
# End of ELF base header

# 01 00 00 00               # ph_type: PT-LOAD = 1
# 00 00 00 00               # ph_offset

00 80 04 08                 # ph_vaddr
00 80 04 08                 # ph_physaddr

B5 00 00 00                 # ph_filesz
B5 00 00 00                 # ph_memsz

01 00 00 00                 # ph_flags: PF-X = 1
01 00 00 00                 # ph_align

#:ELF_text

# Where the ELF Header is going to hit
# Simply jump to _start
# Our main function
# :_start ; (0x804804C)
  58                          ; pop_eax           # Get the number of arguments
  5B                          ; pop_ebx           # Get the program name
  5B                          ; pop_ebx           # Get the actual input name
  31C9                        ; xor_ecx,ecx       # prepare read_only, ecx = 0
  6A 05                       ; push !5           # prepare to set eax to 5
  58                          ; pop_eax           # the syscall number for open()
  99                          ; cdq               # Extra sure, edx = 0
  CD 80                       ; int !0x80         # Now open that damn file
  5B                          ; pop_ebx           # Get the output name
  50                          ; push_eax          # Preserve the file pointer we were given
  66B9 4102                   ; mov_cx, @577      # Prepare file as O_WRONLY|O_CREAT|O_TRUNC
  66BA C001                   ; mov_dx, @448      # Prepare file as RWX for owner only (700 in octal)
  6A 05                       ; push !0x5         # Prepare to set eax to 5
  58                          ; pop_eax           # the syscall number for open()
  CD 80                       ; int !0x80         # Now open that damn file
  99                          ; cdq               # edx = 0 since file descriptor is nonnegative
  42                          ; inc_edx           # edx = 1 (count for read/write)
  97                          ; xchg_eax,edi      # Preserve outfile

#:loop_reset_all ; (0x8048069)
  31ED                        ; xor_ebp,ebp             # ebp = 0 (no prior hex val)

# Comment tracking is done with esi.
# esi is decremented if we hit a
# comment (';' or '#') and reset
# if we hit a new-line.
#:loop_reset_comment ; (0x804806B)
  89D6                        ; mov_esi,edx             # Set no current comment
#:loop_add_comment ; (0x804806D)
  4E                          ; dec_esi
#:loop ; (0x804806E)

  # Read a byte
  5B                          ; pop_ebx                 # Get infile
  89E1                        ; mov_ecx,esp             # Set buffer
  # edx is already set to 1.
  6A 03                       ; push !3
  58                          ; pop_eax                 # Set read syscall in eax
  CD 80                       ; int !0x80               # Do the actual read
  53                          ; push_ebx                # Re-save infile
  85C0                        ; test_eax,eax            # Check what we got
  75 05                       ; jne !cont               # No EOF

  # Exit successfully
  40                          ; inc_eax                 # Set exit syscall in eax
  31DB                        ; xor_ebx,ebx             # Set return success (ebx = 0)
  CD 80                       ; int !0x80               # Exit

#:cont ; (0x8048080)
  8A01                        ; mov_al,[ecx]            # Move prog byte in eax

  # New line check
  3C 0A                       ; cmp_al, !10             # Check new-line
  74 E5                       ; je !loop_reset_comment  # If new-line, end comment handling

  # In comment check
  85F6                        ; test_esi,esi            # Skip byte if we are in a comment
  75 E4                       ; jne !loop

  # Start comment check
  3C 23                       ; cmp_al, !35             # Start of '#' comment
  74 DF                       ; je !loop_add_comment

  3C 3B                       ; cmp_al, !59             # Start of ';' comment
  74 DB                       ; je !loop_add_comment

  # Start of hex str to int
  2C 30                       ; sub_al, !48             # Subtract ascii '0' from al
  2C 0A                       ; sub_al, !10             # Check for value in '0'-'9'
  72 08                       ; jb !write               # We have hex value, write it

  2C 07                       ; sub_al, !7              # Subtract ('A'-'0') from al
  24 DF                       ; and_al, !0xDF           # Remove lower case bit
  3C 07                       ; cmp_al, !7              # Check for value 'A'-'F'
  73 CE                       ; jae !loop               # We don't have hex value ignore it

#:write ; (0x80480A8)
  C1E5 04                     ; shl_ebp, !4             # Shift up existing hex digit
  04 0A                       ; add_al, !10             # Finish converting ascii to raw value
  01C5                        ; add_ebp,eax             # Combine the hex digits

  # Check if this is first digit in hex val
  F7DF                        ; neg_edi                 # Flip sign of edi to indicate we got a digit
  7C C3                       ; jl !loop                # Negative -> first digit, get another one

  # We have both digits in low byte of ebp, good to write
  8929                        ; mov_[ecx],ebp           # Move edge to buffer
  89FB                        ; mov_ebx,edi             # Move outfile to ebx
  B0 04                       ; mov_al, !4              # Set write syscall in eax
  CD 80                       ; int !0x80               # Do the write
  EB B4                       ; jmp !loop_reset_all     # Start a fresh byte
#:ELF_end ; (0x80480B5)
```

Listing B.1. Complete `hex0` x86 assembly source code.

Appendix C

Jam.sh Source Code

In this appendix, we include the POSIX shell source code for the `jam` utility that works for text files. This utility can create self-extracting POSIX shell archives and is used for the packaging of the pnut reproducible builds. The variant of `jam` that works for binary files is not included here for brevity, but works with similar principles (with an additional base64 encoding and decoding step) and is available in the pnut repository [18].

```
#! /bin/sh
# Jam utility: Create self-extracting POSIX shell archives containing text files.
# Example usage:
#   ./jam.sh file1 file2 dir1 dir2

EOF_SEP="EOF3141592653"

# Replace every non-alphanumeric character with _
normalize_name() { # $1: name to normalize
  name="$1"
  normalized_name=""
  while [ -n "$name" ]; do
    char="${name%"${name#?}"}" # Get the first character
    name="${name#?}"           # Remove the first character
    case "$char" in
      [!a-zA-Z0-9]) char="_" ;;
    esac
    normalized_name="${normalized_name}${char}"
  done
}

# POSIX shell implementation of the cat utility
pcat() {
  while IFS= read -r line; do
    printf "%s\n" "$line"
  done
}

```

```
# Add pcat function to output
gen_pcat() {
  pcat << 'EOF'
pcat() {
  while IFS= read -r line; do
    printf "%s\n" "$line"
  done
}

EOF
}

process_file() { # $1: file to process, $2: path
  normalize_name "$1" # Returns in normalized_name variable
  printf "extract_%s() {\n" "$normalized_name"
  printf "  printf \"Extracting %s\\\\n\"\n" "$1"
  printf "  pcat << '%s' > %s\n" "$EOF_SEP" "$1"
  pcat "$1" < "$1"
  printf "%s\n}\n\n" "$EOF_SEP"
  printf "extract_%s\n\n" "$normalized_name"
}

process_dir() {
  IFS=" "
  for file in "$1"/*; do
    if [ -f "$file" ]; then
      process_file "$file"
    elif [ -d "$file" ]; then
      process_dir "$file"
    fi
  done
}

# Generate header
printf "#! /bin/sh\n\n"
gen_pcat

# For each file/directory argument, add its content to the archive
for arg in "$@"; do
  if [ -f "$arg" ]; then
    process_file "$arg"
  elif [ -d "$arg" ]; then
    process_dir "$arg"
  fi
done
```

Listing C.1. Source code of the text-only `jam` utility.

Appendix D

Sift.sh Source Code

In this appendix, we include the POSIX shell source code for the `sift.sh` utility. This utility can extract files from self-extracting POSIX shell archives, without executing the archive script. Because of its short length, the script can be typed by hand to audit jam archives.

```sh
#! /bin/sh
# sift.sh: Extract files in a jammed archive without running it.
# Example usage:
#   ./sift.sh pnut.c < jammed.sh

set -e -u
error() { printf "Error: %s\n" "$1" >&2; exit 1; }

if [ $# -lt 1 ]; then error "Usage: $0 file_path < archive_file"; fi
file="$1"
EOF_MARKER="EOF3141592653"

# Read until we find a line containing "> $file"
while IFS= read -r line; do
  [ "$line" != "${line#*> $file}" ] && break
done

# Then read lines until we find the EOF_MARKER line, printing them out
while IFS= read -r line; do
  [ "$line" = "$EOF_MARKER" ] && break
  printf "%s\n" "$line"
done

# If we reached EOF without finding the marker, the file was not found
if [ "$line" != "$EOF_MARKER" ]; then
  error "File '$file' not found in archive."
fi
```

Listing D.1. Source code of the `sift.sh` utility.

Appendix E

More.sh Source Code

In this appendix, we include the POSIX shell source code for the `more` [32] pager. This simple pager can be used to view text files in the terminal. Because of its short length, the script can be typed by hand to audit files extracted from jam archives.

```sh
#! /bin/sh
# more.sh: Basic pager that displays text one screen (24 lines) at a time.
# Example usage:
#   ./more.sh some_file

set -e -u
error() { printf "Error: %s\n" "$1" >&2; exit 1; }

if [ $# -lt 1 ]; then error "Usage: $0 file_path"; fi

LINES_PER_PAGE=24

block_on_user_input() {
  IFS= read -r user_input                     # Wait for user input on stdin
  if [ -n "$user_input" ]; then exit 0; fi # Exit on any non-empty input
}

exec 3< "$1" # Open file for reading on fd 3
line_count=0
while IFS= read -r line <&3; do
  printf "%s\n" "$line"
  : $(( line_count += 1 ))
  if [ $line_count -ge $LINES_PER_PAGE ]; then
    block_on_user_input
    line_count=0
  fi
done
```

Listing E.1. Source code of the `more.sh` utility.