Bringing RISC-V to Guix's bootstrap

What's done and what we need to do

Ekaitz Zárraga

Who I am

- Telecommunication engineer (EEE equivalent)
- Freelance engineer/programmer at ElenQ.Tech[1]
- Guix user and contributor
- You might remember me from my talk last year: *“A year of RISC-V adventures: embracing chaos in your software journey”*

[1] https://elenq.tech

Intro

- Last year I asked NlNet for a grant[2].
- I wanted to push the bootstrapping effort for RISC-V, and they funded me to do so.
- In this talk I'm going to introduce what I did[3], what I think it's more or less done and what's missing.

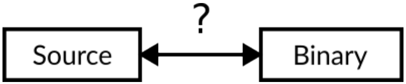
[2]https://nlnet.nl/project/GNUMes-RISCV/index.html

[3]Read the longer version here: https://ekaitz.elenq.tech/tag/bootstrapping-gcc-in-risc-v.html

Intro to bootstrapping

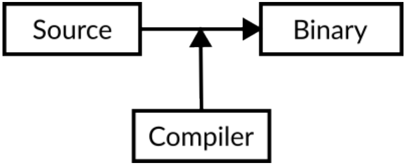
Free software is not enough

We love Free Software because it helps us audit our programs.



But do we know if the source code we read actually maps to the binary we are executing? **Not really**

Reproducibility

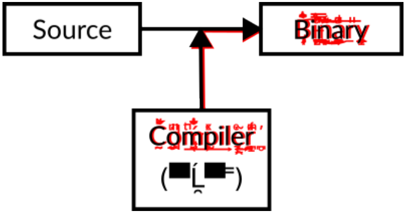


The relation is one-way: the compiler is in the middle

In Guix we have **reproducibility**, so we can make sure some inputs (the source, the compiler and the environment where it runs) always produce the same outputs.

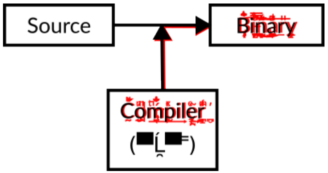
We can challenge the binaries, so nobody will give us a malicious binary.

Trusting trust



But what if the bad actor is not a person but a program?

Trusting trust



But what if the bad actor is not a person but a program?

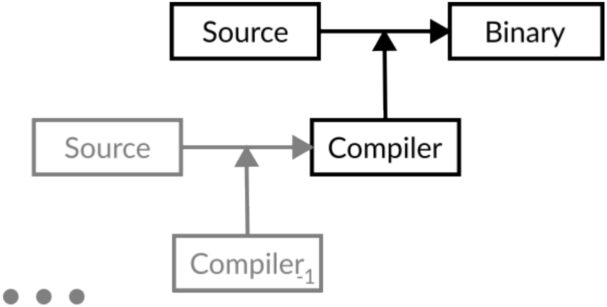
Reproducibility here will only make sure we generate the same **corrupt** binary.

This kind of attack can be done in real life[4].

[4]https://niconiconi.neocities.org/posts/ken-thompson-really-did-launch-his-trusting-trust-trojan-attack-in-real-life/

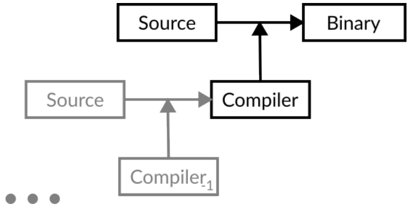
Recursive problem, recursive solution

The compiler is a program too. **This issue is recursive**: a corrupt compiler could corrupt it's output compiler!



What's the exit point?

Recursive problem, recursive solution



What’s the exit point?

If we could have a compiler that we can make sure its output is not corrupt (**but how?**), we could make sure all the chain is correct.

In practice

GNU+Linux distributions often rely in many prebuilt binaries: Bash, GCC, Coreutils, Python…

Some distributions like Guix are interested on reducing the amount of binaries they have to trust.

We can compile most of **The World** from source using a powerful compiler (GCC FTW). But we can’t use a pre-built compiler (remember the previous slides?)

The key: Who is compiling the compiler?

In practice - II

Let’s try with GCC:

0. The World (requires a modern GCC)
1. Modern GCC (requires ISO C++11 compiler)

In practice - II

Let’s try with GCC:

0. The World (requires a modern GCC)
1. Modern GCC (requires ISO C++11 compiler)
2. GCC 11 (requires ISO C++98 compiler)

In practice - II

Let’s try with GCC:

0. The World (requires a modern GCC)
1. Modern GCC (requires ISO C++11 compiler)
2. GCC 11 (requires ISO C++98 compiler)
3. GCC 4.8 (requires ISO C89 compiler)

In practice - II

Let’s try with GCC:

0. The World (requires a modern GCC)
1. Modern GCC (requires ISO C++11 compiler)
2. GCC 11 (requires ISO C++98 compiler)
3. GCC 4.8 (requires ISO C89 compiler)
4. GCC 3.4 (requires K&R compiler)

In practice - II

Let’s try with GCC:

0. The World (requires a modern GCC)
1. Modern GCC (requires ISO C++11 compiler)
2. GCC 11 (requires ISO C++98 compiler)
3. GCC 4.8 (requires ISO C89 compiler)
4. GCC 3.4 (requires K&R compiler)
5. …

In practice - II

Let’s try with GCC:

⓿ The World (requires a modern GCC)

❶ Modern GCC (requires ISO C++11 compiler)

❷ GCC 11 (requires ISO C++98 compiler)

❸ GCC 4.8 (requires ISO C89 compiler)

❹ GCC 3.4 (requires K&R compiler)

❺ …

I didn’t mention libraries here, that’s also a lot of fun

Guix's bootstrapping

0. The World
1. Modern GCC
2. GCC 7.5
3. GCC 4.6.4
4. GCC 2.95
5. TinyCC
   - Bootstrappable TinyCC
6. GNU Mes
7. Stage0-POSIX => **SOURCE CODE**

GNU Mes

GNU Mes is a Scheme interpreter and C compiler for bootstrapping the GNU System. Since version 0.22 it has again helped to halve the size of opaque, uninspectable binary seeds that are currently being used in the Further Reduced Binary Seed bootstrap of GNU Guix. ***The final goal is to help create a full-source bootstrap as part of the bootstrappable builds effort for UNIX-like operating systems****.*

The Scheme interpreter is written in ~5,000 LOC of simple C, and the C compiler written in Scheme and these are mutual self-hosting. Mes can now be bootstrapped from M2-Planet and Mescc-Tools.

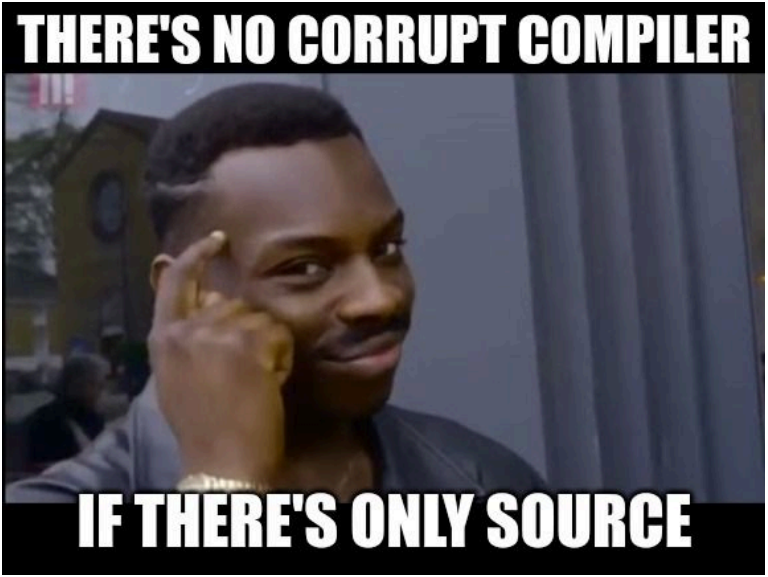
https://www.gnu.org/software/mes/

Stage0-POSIX

> *It bootstraps all these from a single 256 byte seed (which you will find in the folder bootstrap-seeds). **The ultimate goal is for this to bootstrap all the way up to GCC**.*
>
> *There is only one "missing" part that is not bootstrappable from the hex0 seed: a kernel. This issue is not yet solved and at the moment the kernel is trusted.*

https://github.com/oriansj/stage0-posix

Boostrapping - wrapping up



RISC-V support

Guix’s bootstrapping - RISC-V support

⓿ The World	• N/A
❶ Modern GCC	• YES
❷ GCC 7.5	• YES
❸ GCC 4.6.4	• NO
❹ GCC 2.95	• NO
❺ TinyCC	• YES
• Bootstrappable TinyCC	• NO
❻ GNU Mes	• PARTIAL
❼ Stage0-POSIX	• YES

Guix's bootstrapping - RISC-V support *SPOILER*

⓿ The World	N/A
❶ Modern GCC	YES
❷ GCC 7.5	YES
❸ GCC 4.6.4	~~NO~~ *I backported this*
❹ GCC 2.95	~~NO~~ *We will remove it*
❺ TinyCC	YES
• Bootstrappable TinyCC	• ~~NO~~ *I backported this*
❻ GNU Mes	PARTIAL
❼ Stage0-POSIX	YES *I made some of this*

What I did

GCC

GCC uses a Davidson-Fraser model. Meaning that it uses an intermediate language that is machine dependant: RTL (Register Transfer Language).

```
HLL -> GIMPLE -> RTL -> OPTIMIZATIONS -> RTL -> ASSEMBLY
```

GCC is only a coordinator: it calls `as` and `ld` from binutils as the assembler and linker.

- `GIMPLE -> RTL`: is done using identifiers. The GIMPLE nodes match insn identifiers.
- `RTL -> OPTIMIZATIONS`: RTL matches the RTL templates we write in the backend part of GCC. Those can be expanded to other RTL expressions.
- `RTL -> ASSEMBLY`: The expanded RTL expressions are matched against RTL templates that also describe their equivalent in assembly and assembly is generated from them.

GCC

RTL templates are written in LISP in machine descriptor files (`*.md`), they look like this:

```
(define_insn
  "adddi3"             ;; Identifier

  ;; The behavior of the instruction
  [(set (match_operand:DI 0 "register_operand" "=r,r")
        (plus:DI (match_operand:DI 1 "register_operand" "r,r")
                 (match_operand:DI 2 "arith_operand"    "r,I")))
   ]

  "TARGET_64BIT"       ;; Predicate to test
  "add\t%0,%1,%2"      ;; Assembly output template

  ;; Attributes
  [(set_attr "type" "arith")
   (set_attr "mode" "DI")])
```

GCC

Apart from that GCC needs tons of other definitions in order to get another target:

- Target description macros and functions
- Libraries like libgcc and many others

GCC - What I did

Cherry picked the RISC-V support from GCC 7.5 to GCC 4.6.4

1. There were missing insns => Used older ones that were equivalent.
2. Some RTL constructs (`int_iterator`) didn't exist in 4.6.4 => Expanded the iterator by hand.
3. There were missing predicates => Copied them.
4. The internal GCC API moved from C to C++ in the meantime => I had to convert the code from using a class to the older interface.
5. Memory barriers didn't exist back then => Always introduce a `fence` to make sure code is correct.
6. `libgcc` is a mess => Play around until it works

TL;DR: Touch everything until it works.

GCC - What I did

Finally I managed to make a GCC 4.6.4 that is able to generate RISC-V binary.

See the blog for a more detailed description of the changes:

- https://ekaitz.elenq.tech/bootstrapGcc3.html
- https://ekaitz.elenq.tech/bootstrapGcc4.html

Bootstrappable TinyCC

TinyCC has RISC-V support but it’s not boostrappable using GNU Mes.

The bootstrappable fork of TinyCC GNU Mes uses is old => Backport again.

Bootstrappable TinyCC - What I did

Copy the relevant files from the upstream TinyCC and:

0. Prepare a reproducible way to build the Bootstrappable TinyCC.
1. Just read the code and make it match.
   SURPRISE: The code is really hard to read… But I eventually managed to make it work.
2. Some core code was needed for the backend to work => Remove it! It was only some optimization code!

More detailed description of the changes:

- https://ekaitz.elenq.tech/bootstrapGcc6.html

Bootstrappable TinyCC - What I did

OPTIMIZED VERSION

```
0000000000000000 <main>:
   0:   fd010113    addi    sp,sp,-48
   4:   02113423    sd  ra,40(sp)
   8:   02813023    sd  s0,32(sp)
   c:   03010413    addi    s0,sp,48
  10:   00000013    nop
  14:   fea43423    sd  a0,-24(s0)
  18:   feb43023    sd  a1,-32(s0)
  1c:   0130051b    addiw   a0,zero,19
  20:   fca42e23    sw  a0,-36(s0)
  24:   05a0051b    addiw   a0,zero,90
  28:   fca42c23    sw  a0,-40(s0)
  2c:   fdc42503    lw  a0,-36(s0)
  30:   00051463    bnez    a0,38 <main+0x38>
  34:   0180006f    j   4c <main+0x4c>
  38:   fd842503    lw  a0,-40(s0)
  3c:   00051463    bnez    a0,44 <main+0x44>
  40:   00c0006f    j   4c <main+0x4c>
  44:   0010051b    addiw   a0,zero,1
  48:   0100006f    j   58 <main+0x58>
  4c:   00008537    lui a0,0x8
  50:   7005051b    addiw   a0,a0,1792
  54:   00000033    add zero,zero,zero
  58:   02813083    ld  ra,40(sp)
  5c:   02013403    ld  s0,32(sp)
  60:   03010113    addi    sp,sp,48
  64:   00008067    ret
```

UNOPTIMIZED VERSION

```
0000000000000000 <main>:
   0: fd010113    addi    sp,sp,-48
   4: 02113423    sd  ra,40(sp)
   8: 02813023    sd  s0,32(sp)
   c: 03010413    addi    s0,sp,48
  10: 00000013    nop
  14: fea43423    sd  a0,-24(s0)
  18: feb43023    sd  a1,-32(s0)
  1c: 0130051b    addiw   a0,zero,19
  20: fca42e23    sw  a0,-36(s0)
  24: 05a0051b    addiw   a0,zero,90
  28: fca42c23    sw  a0,-40(s0)
  2c: fdc42503    lw  a0,-36(s0)
  30: 00051463    bnez    a0,38 <main+0x38>
  34: 01c0006f    j   50 <main+0x50>
  38: fd842503    lw  a0,-40(s0)
  3c: 00051463    bnez    a0,44 <main+0x44>
  40: 0100006f    j   50 <main+0x50>
  44: 0010051b    addiw   a0,zero,1
  48: 0140006f    j   5c <main+0x5c>
  4c: 0100006f    j   5c <main+0x5c>
  50: 00008537    lui a0,0x8
  54: 7005051b    addiw   a0,a0,1792
  58: 00000033    add zero,zero,zero
  5c: 02813083    ld  ra,40(sp)
  60: 02013403    ld  s0,32(sp)
  64: 03010113    addi    sp,sp,48
  68: 00008067    ret
```

What needs to be done

GCC - What needs to be done

- Properly package the GCC-4.6.4 to include C++ support, and fix all the libraries that might be missing.
- Build the backported GCC-4.6.4 using TinyCC
- Build GCC-7.5 using the backported GCC-4.6.4

TinyCC - What needs to be done

- Build the bootstrappable TinyCC using GNU Mes
- Decide if we need the upstream TinyCC to build GCC or not
  - If we do: build the upstream TinyCC with the bootstrappable TinyCC

GNU Mes - What needs to be done

- Review the current RISC-V support and prepare it to be merged

Guix - What needs to be done

- Describe the whole compiler compiler chain in the bootstrapping packages so everyone can benefit from it

Extra

Do it all in real hardware

Last words

Last words

There’s still a lot of work to be done, most of it being the integration of the work I already did in the past thanks to NlNet.

The future is bright though. We are probably going to get more funds from NlNet to involve more people on this.

Wanna join?

Contact and take part

- Email me: ekaitz@elenq.tech[5]
- Relevant IRC channels: `#bootstrappable`, `#guix`, `#guix-risc-v`

[5] mailto:ekaitz@elenq.tech

Thank you