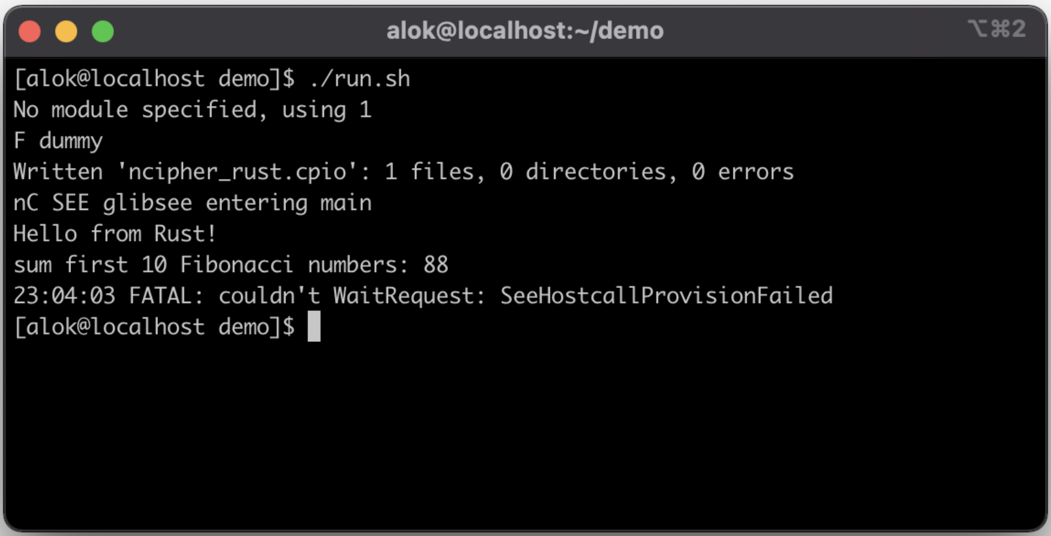
# Rust on nCipher 🦀🛡️

Alok Menghrajani ~~April 1st~~ March 31st ,2021

Not a Rust expert! This document is serious, kind of.



```
[alok@localhost demo]$ ./run.sh
No module specified, using 1
F dummy
Written 'ncipher_rust.cpio': 1 files, 0 directories, 0 errors
nC SEE glibsee entering main
Hello from Rust!
sum first 10 Fibonacci numbers: 88
23:04:03 FATAL: couldn't WaitRequest: SeeHostcallProvisionFailed
[alok@localhost demo]$
```

## Context

Square uses nCipher HSMs to protect sensitive cryptographic keys. The device has a PCIe form factor (other variants such as usb or network do exist). The device supports basic cryptographic primitives, such as RSA and AES. When we need to perform more advanced cryptographic operations, we typically write custom code in C, for example:

- Subzero, which implements a Bitcoin offline wallet as well as custom business logic rules around authorizing transactions.

The C code uses a feature called CodeSafe and runs in a special enclave, Secure Execution Environment (SEE), on the nCipher. The nCipher vendor supplies a gcc cross-compiler toolchain and pre-compiled libraries (such as libc).

**The purpose of this document is to demonstrate that we can write CodeSafe code in Rust.**

Using Rust instead of C provides several advantages:

1. Rust is a cool language. C is uncool.
2. Rust uses linear types. Linear types are cool. C uses explicit memory management. Explicit memory management is uncool.
3. Rust has a package management system and a rich ecosystem of libraries. Rust's cryptographic libraries are trusted. In C, I have previously used trezor-crypto, which has some nice things but isn't widely vetted.
4. Rust's tooling is much better than C fractal of broken tools. Compiling/cross-compiling Rust code is magic.

## A short overview of compiler design (by not a compiler expert)

Compilers are typically split in two parts: the front-end and the back-end.

The compiler's front-end takes source code (typically text files), uses a parser/grammar and generates an intermediate representation (IR), typically a tree. The front-end is responsible for syntax checking, type checking, etc. In Rust's case, the front-end implements the borrow checker, which is core to the linear type system. Real world compilers will use multiple different intermediate representations, IR here refers to the last intermediate representation, which is typically a stable format.

The compiler's back-end takes the IR and spits out code (typically assembly but not necessarily) for the desired target platform. The backend is responsible for optimizing the IR. Again, real world compilers will have backends with multiple layers, might use different representations such as flow graphs, etc.

The clean front/back split for compilers enables supporting N different languages x M different target platforms by only implementing N front-ends + M back-ends (instead of N x M different compilers). Traditionally, the input languages were C and C++ with various different targets.

For example, Rust uses LLVM as one of the backends. The Rust designers targeted LLVM's IR and got all of LLVM's optimizations for free, in addition to being able to run Rust on a wide selection of target platforms.

## nCipher Secure Execution Environment (SEE)

Here is what I know about the target environment. Some of this is documented. Some of it is inferred:

- nCipher communicates with the host machine using a kernel driver and a user space process.
- Freescale PowerPC e5500 processor.

- Either 32-bit processor or 32-bit operating system. In any case, I have never gotten 64-bit code to run on the nCipher.
- Big endian.
- Some old Linux kernel or fork thereof. Some system calls are wrapped and end up communicating with the host server (e.g. stdio and network calls are redirected to the host).

## Possible approaches

I explored various approaches:

- Cross-compiling a Rust application using xargo or cross. In theory, this should work, but I had a hard time getting LLVM's linker to link with our vendor's pre-compiled libraries.
- Tell Rust to emit LLVM IR and then use LLVM IR to target SEE. This might sound similar to the above approach, but gives an opportunity to debug things more efficiently (e.g. easier to try different LLVM versions).
- Tell Rust to emit LLVM IR. Use LLVM-CBE to turn the IR into C code. Use the vendor's gcc-based cross-compiler toolchain to compile the C code. This might sound crazy, but almost worked. The main issue with this kind of approach is making sure the LLVM IR is complete (grabbing the dependencies recursively).
- Use mrustc, an experimental Rust compiler which emits C code instead of LLVM IR.
- Cross-compile a Rust application to WebAssembly. Use a decompiler to turn the wasm code into C. Use the vendor's gcc-based cross-compiler toolchain.

In all these approaches, I was trying to compile an entire Rust application and I wasn't successful. Debugging failures is hard, because in some cases, the code compiles and loads but silently fails to run.

I then tried a different approach:

- Compile the Rust code as a library, call the Rust code from C. Link with the vendor supplied gcc-based toolchain.

## Try it out

I have a very simple piece of Rust code which runs on the nCipher. This is just a proof-of-concept, it computes the sum of the first 10 Fibonacci numbers. I'm not dealing with exceptions, a panic simply aborts. There's also no interaction with the device itself beyond writing to stdout. Calling nCipher's API shouldn't be any different than calling printf though, since it's just an external C function from Rust's point of view.

Compile the Rust code with:

1. Install Docker if you don't already have it.
2. git clone https://github.com/alokmenghrajani/ncipher_rust.git
3. Download CodeSafe-linux64-dev-12.50.2.iso and move it inside codesafe/.
4. ./codesafe/build.sh
5. ./rust/build.sh
6. docker run --rm -it -v $PWD/demo:/demo ncipher-rust /demo/compile.sh

Run on a host with an attached nCipher HSM:

1. scp -r demo <host>:~/
2. ssh <host>
3. $ cd ./demo
4. $ ./run.sh