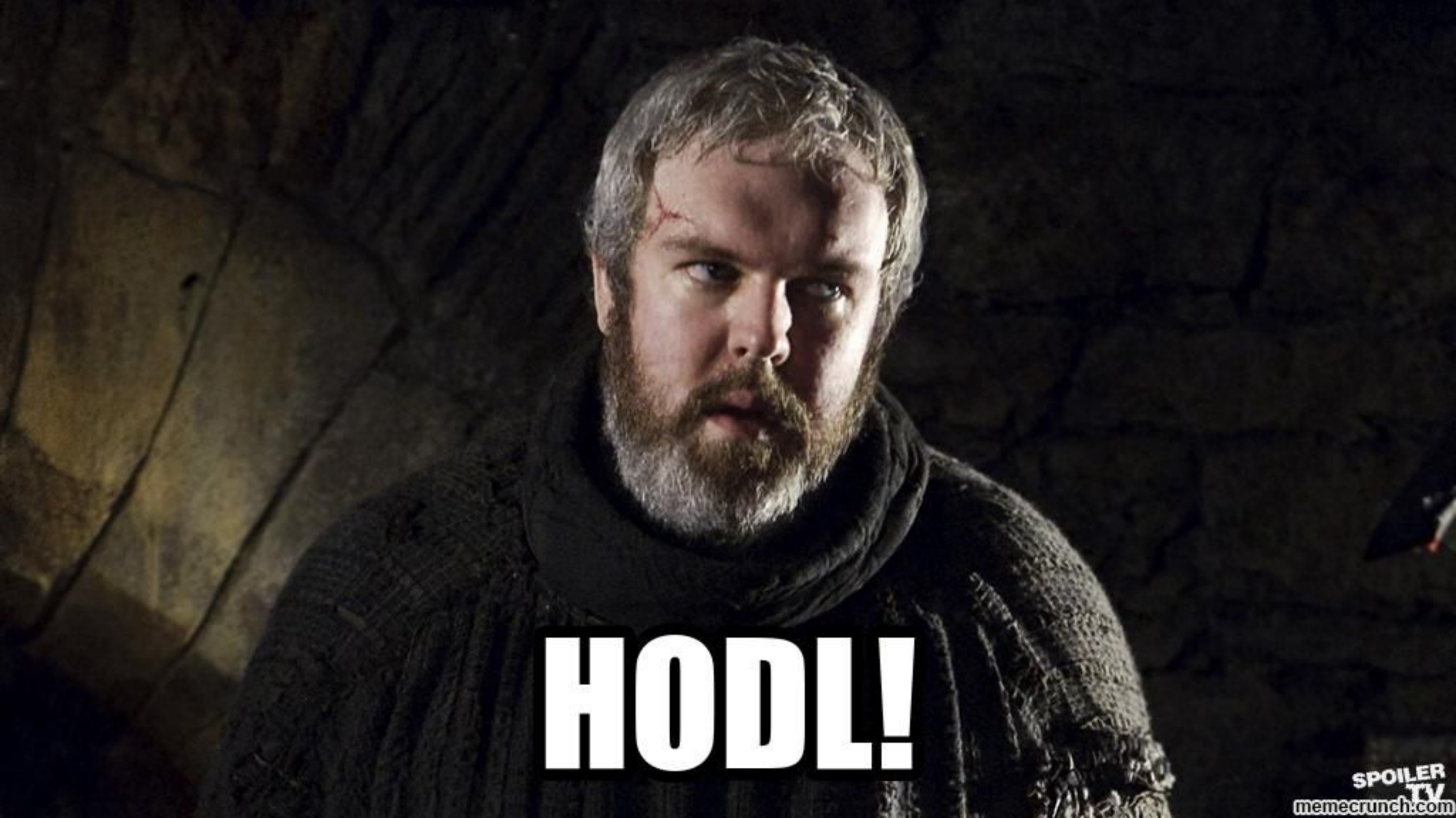
Get Paid in

bitcoin

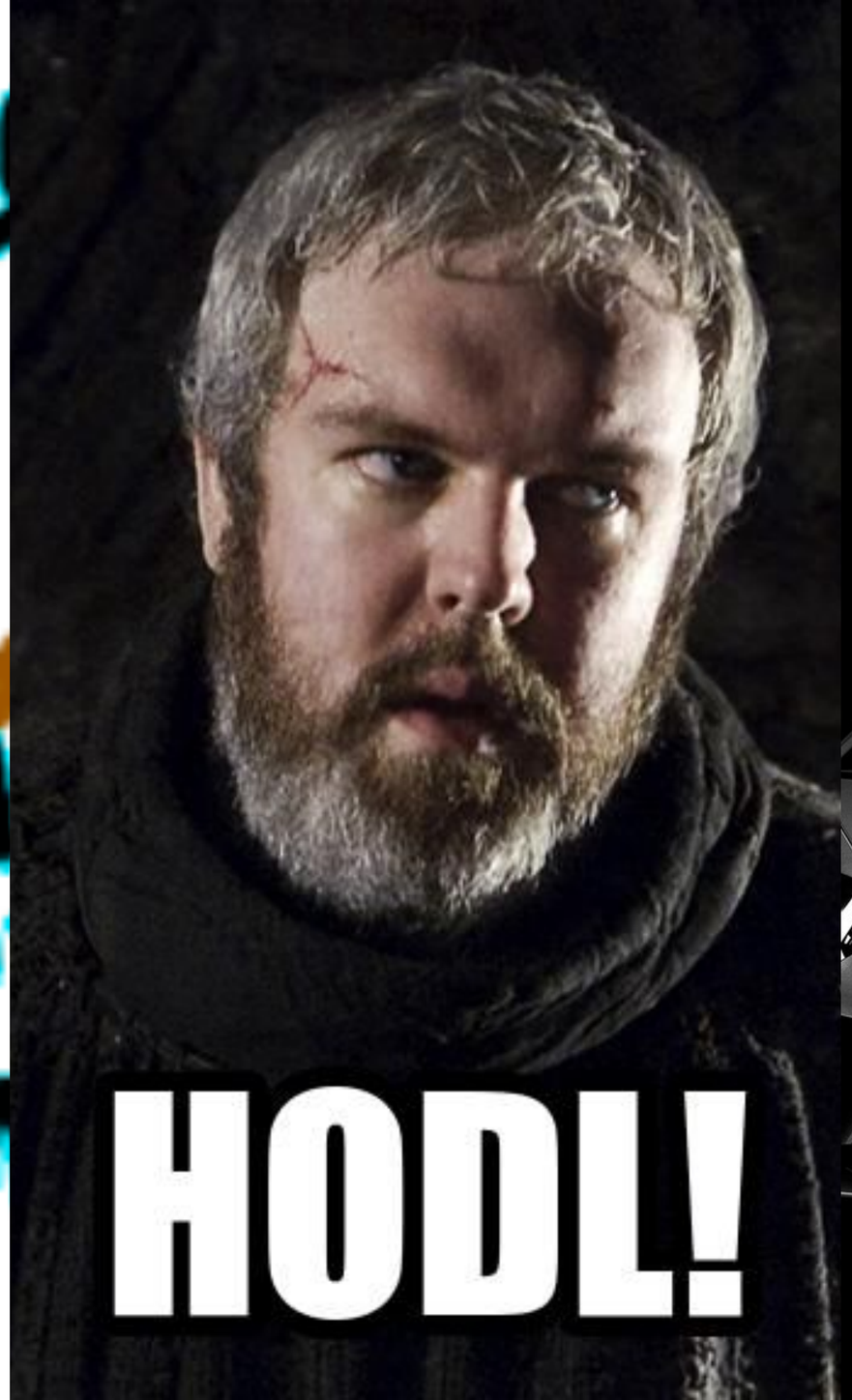

BITCOIN
MAGIC
internet
money


HODL!
SPOILER TV
memecrunch.com


BITCOIN · DIGITAL · DECENTRALIZED · PEER TO PEER
1 TROY OZ .999 FINE COPPER
BULLION MONETARY METALS

bitcoin


Bi To


HODL!


Proud to Accept
bitcoin

1FRvnTotNzKA4JLTddVE5KuSVy8oH2Fq4o

ExCrypto

https://github.com/ntrepid8/ex_crypto

ExCrypto

https://github.com/ntrepid8/ex_crypto

RSA Ex

https://github.com/anoskov/rsa-ex

Commits on Jan 18, 2018

Add formatter and format code
paulfioravanti committed 4 hours ago

Verified | bd1c57b

Minor test code refactor
paulfioravanti committed 7 hours ago

71

Run mix format over codebase
paulfioravanti committed 7 hours ago

b4

Update project to use Elixir 1.6, and comment out use of Dogma for no...

...

paulfioravanti committed 7 hours ago

67

This commit was signed with a **verified signature**.

paulfioravanti
Paul Fioravanti

GPG key ID: 0F6AF064085A6D9C
Learn about signing commits


ut

Generate Bitcoin Address

We need:

A private key

We need:

Bitcoin public key
(from private key)

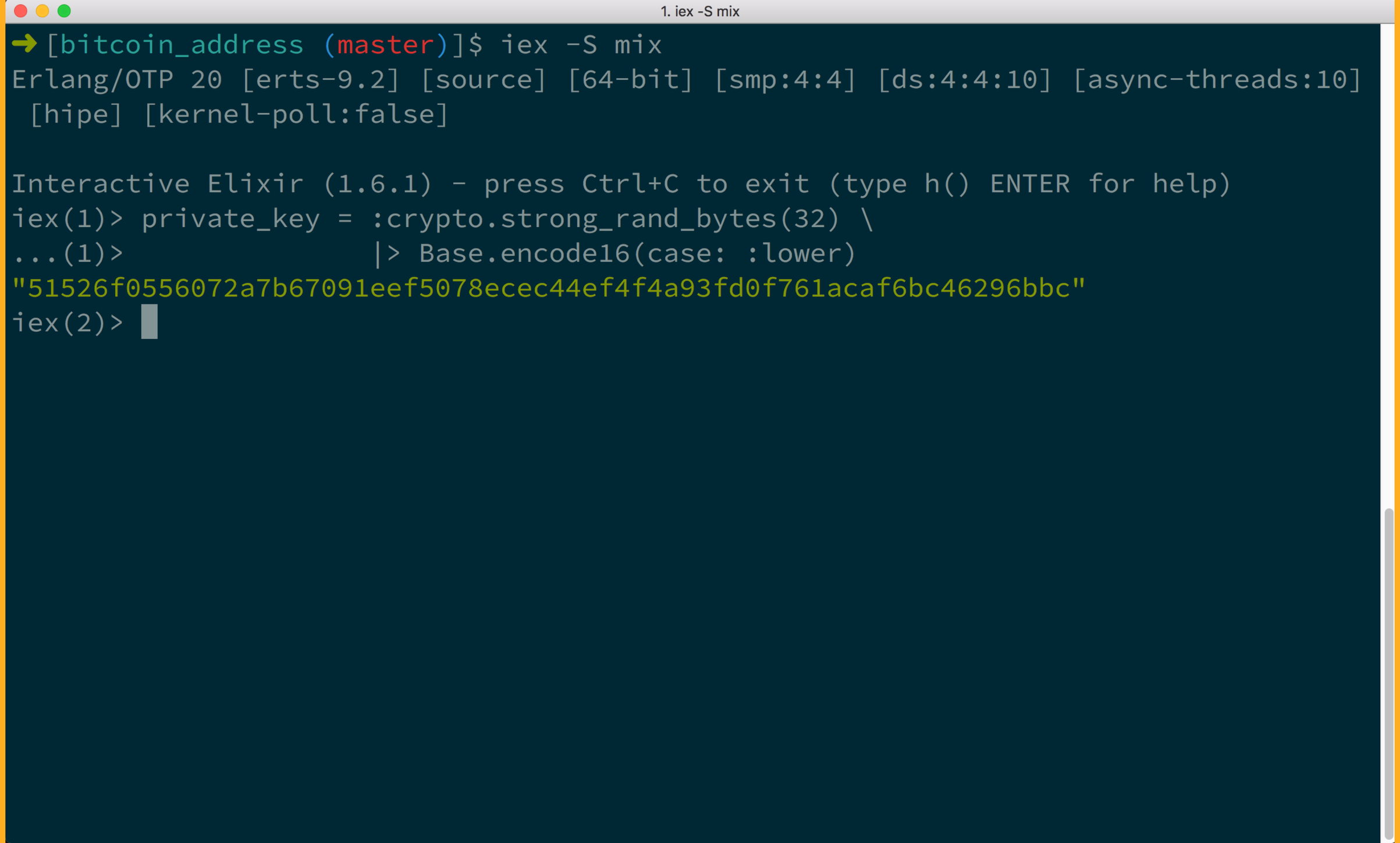
We need:

Bitcoin address

(from Bitcoin public key)

Checklist

- Private Key
- Bitcoin Public Key
- Bitcoin Address


1. iex -S mix
➜ [bitcoin_address (master)]$ iex -S mix
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
[hipe] [kernel-poll:false]
Interactive Elixir (1.6.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> private_key = :crypto.strong_rand_bytes(32) \
...(1)> |> Base.encode16(case: :lower)
"51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
iex(2)>

Checklist

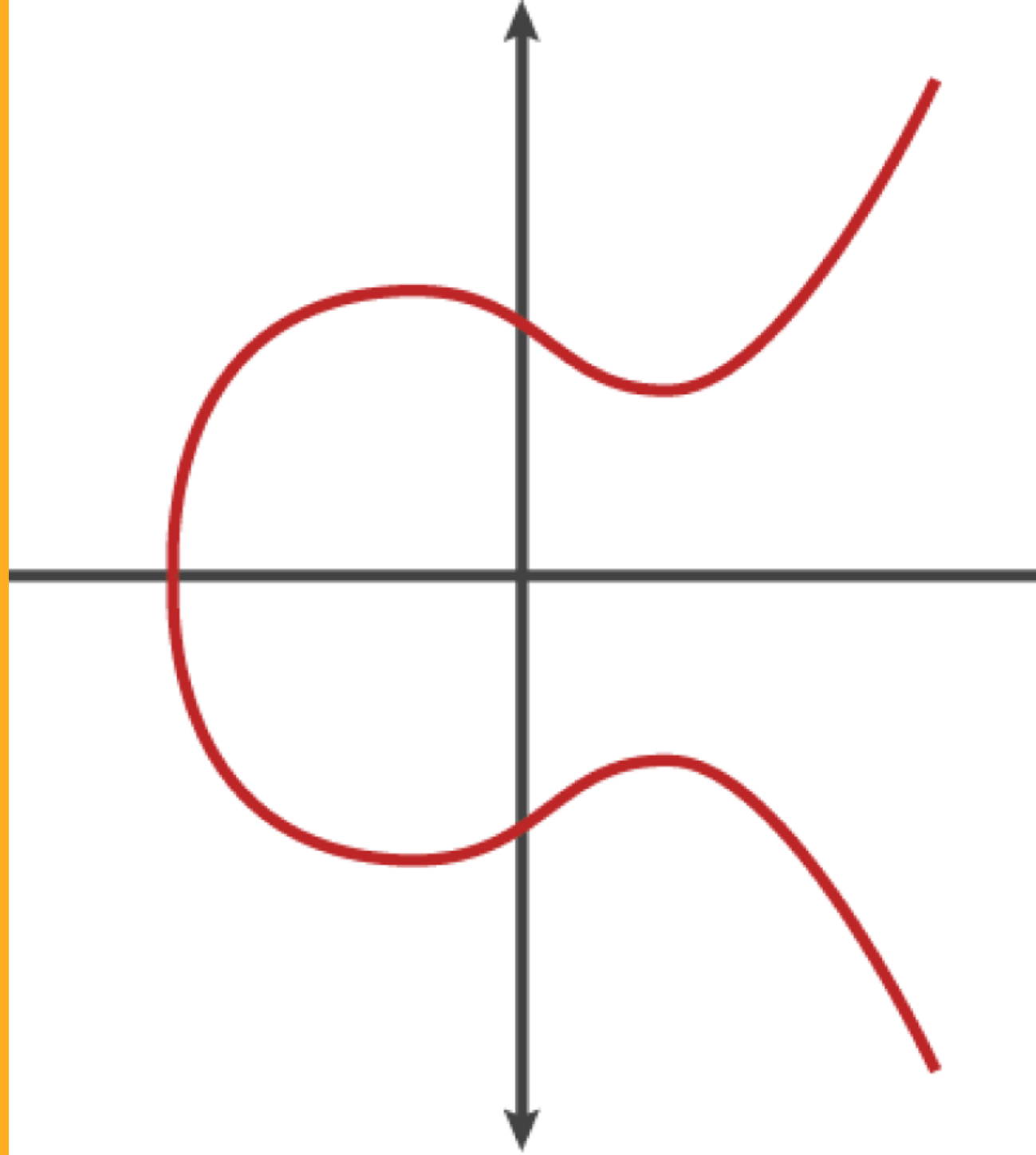
- Private Key ✅
- Bitcoin Public Key
- Bitcoin Address

Bitcoin

Public Key

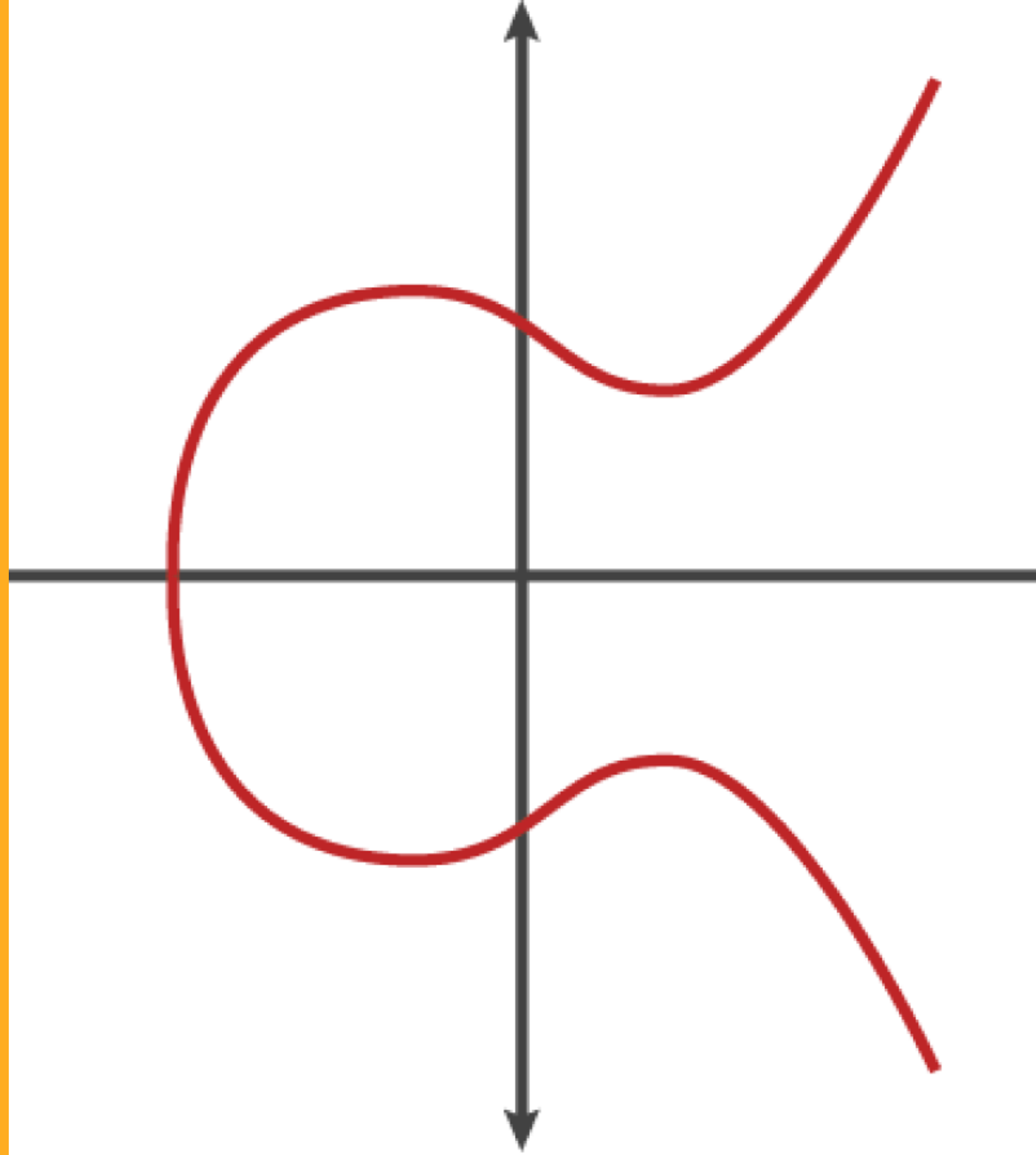
Elliptic Curve Digital Signature Algorithm

@paulfioravanti



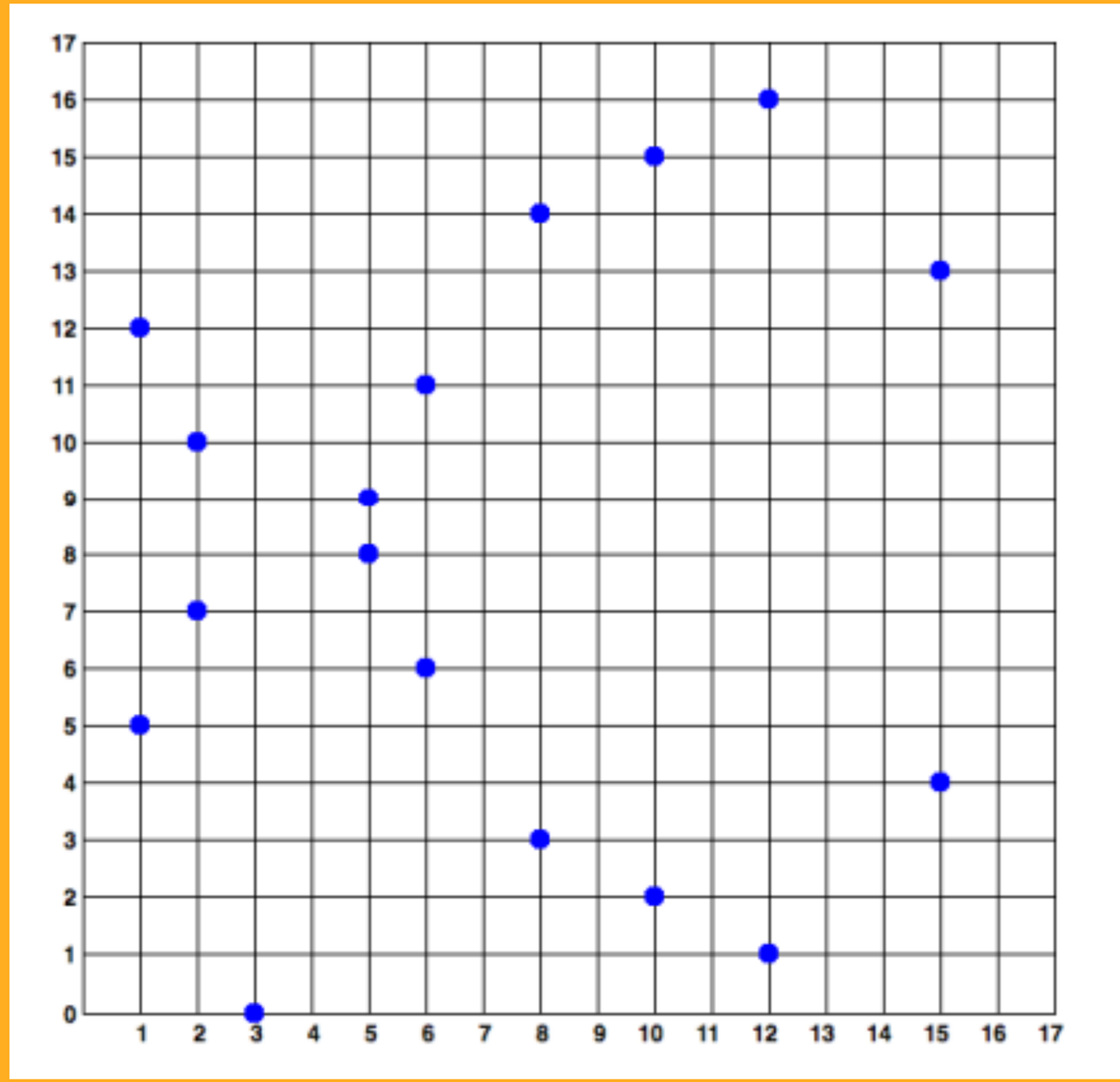
Elliptic Curve Digital Signature Algorithm

@paulfioravanti



ECDSA

secp256k1



@paulfioravanti

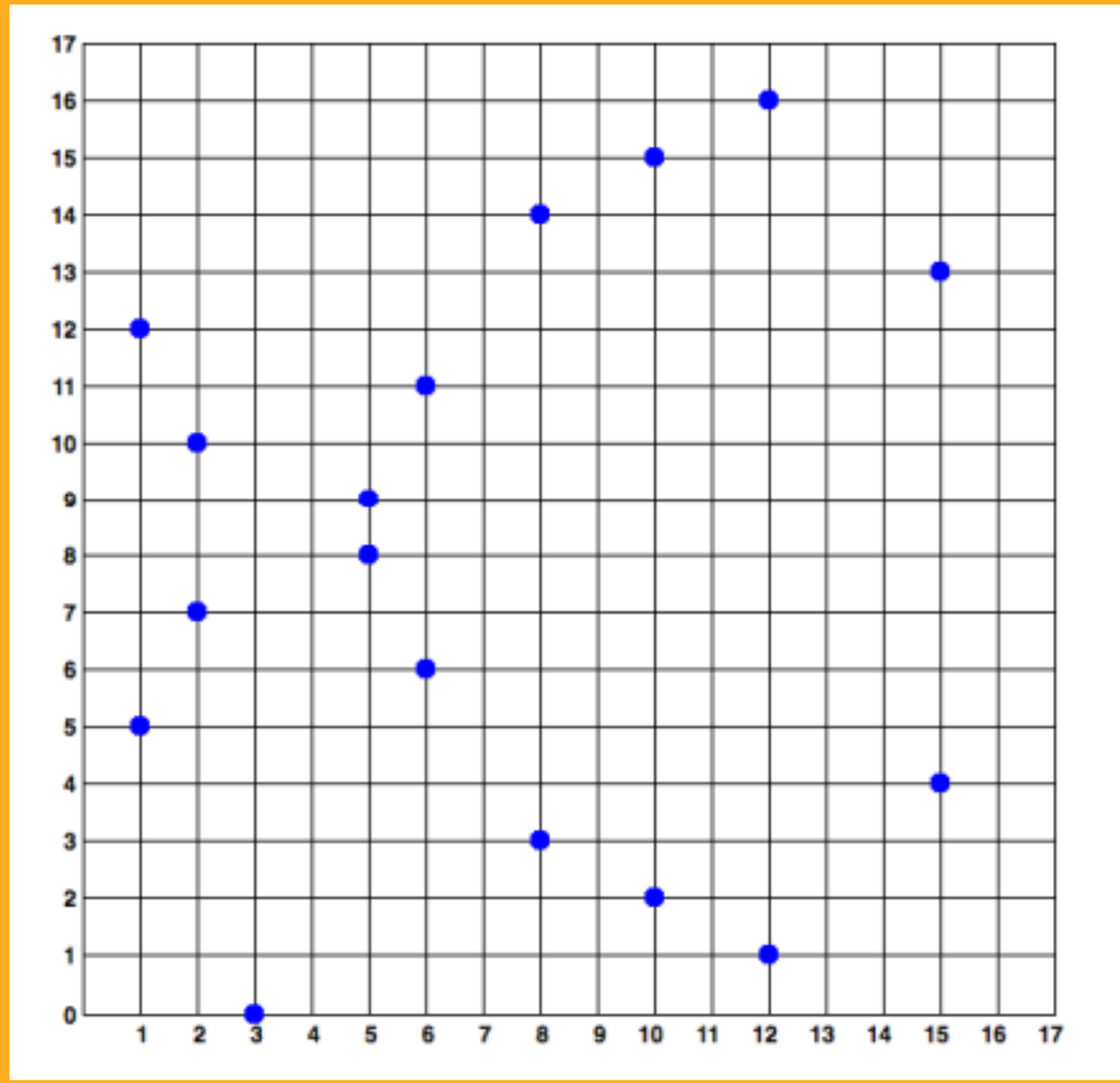
ECDSA

secp256k1

{ x , y }

coordinate on the

Elliptic Curve



@paulfioravanti

Anyway...

bitcoin-elixir

— https://github.com/comboy/bitcoin-elixir

bitcoin-elixir

— https://github.com/comboy/bitcoin-elixir

Bitcoin-Ex

— https://github.com/justinlynn/bitcoin-ex


ut

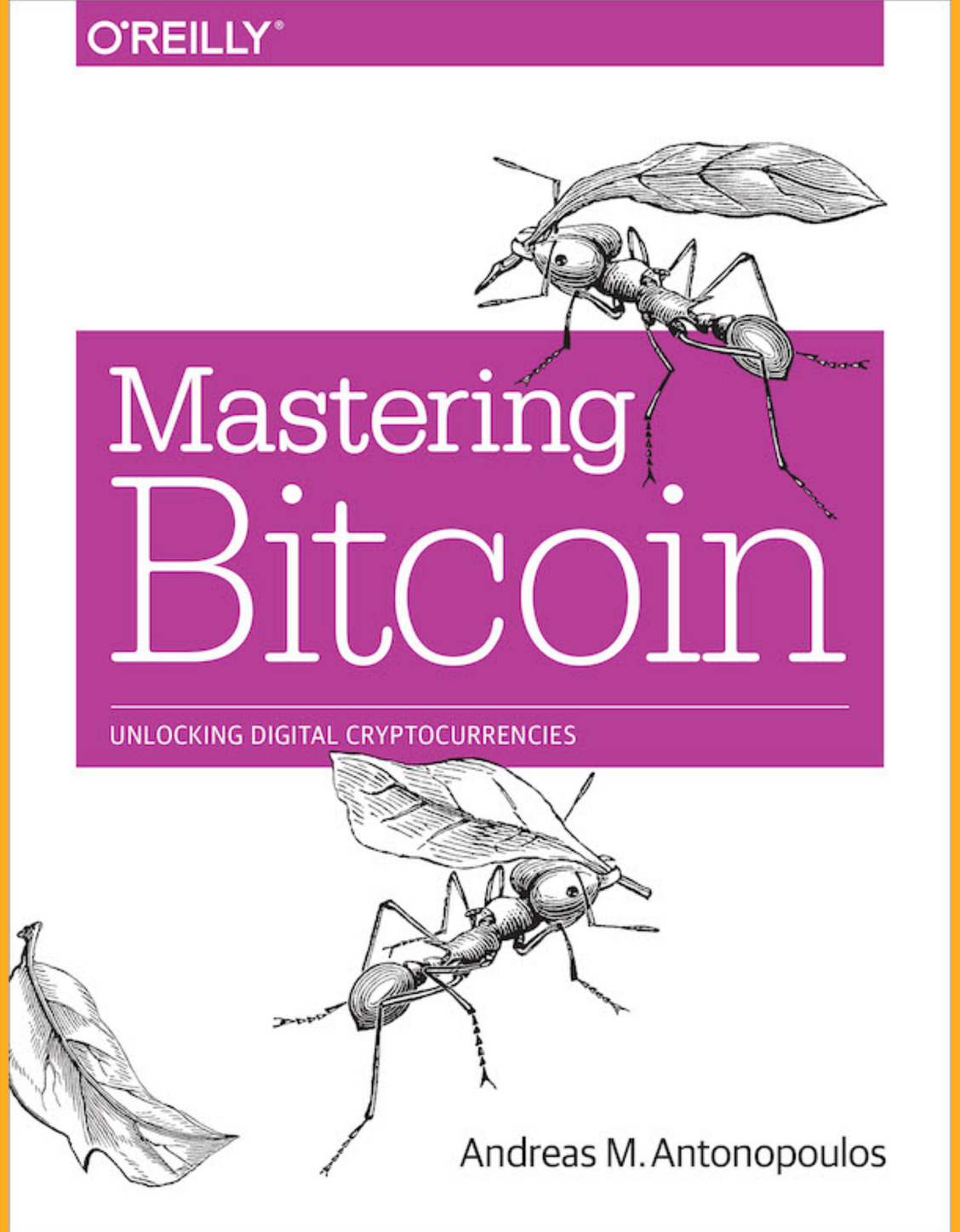
Checklist

- Private Key ✅
- Bitcoin Public Key ❌
- Bitcoin Address ✋

NOW WHAT?

NOPE!


ut


O'REILLY
Mastering
Bitcoin
UNLOCKING DIGITAL CRYPTOCURRENCIES
Andreas M. Antonopoulos


BITCOINERS BE LIKE
I'M IN IT FOR
THE TECHNOLOGY
imgflip.com
©CEN

New project!

```
$ mix new bitcoin_address
$ cd bitcoin_address
$ mix deps.get
```

Pybitcointools

— https://github.com/vbuterin/pybitcointools

```
$ pip install bitcoin
```

Python Code

priv/bitcoin_address.py

bitcoin_address.py

```
import bitcoin

def create_bitcoin_public_key(private_key):
  decoded_private_key =
    bitcoin.decode_privkey(private_key, "hex")
  public_key =
    bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
  (public_key_x, public_key_y) = public_key
  compressed_prefix = "02" if (public_key_y % 2) == 0 else "03"
  return compressed_prefix + bitcoin.encode(public_key_x, 16, 64)
```

bitcoin_address.py

```
import bitcoin

def create_bitcoin_public_key(private_key):
  decoded_private_key =
    bitcoin.decode_privkey(private_key, "hex")
  public_key =
    bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
  (public_key_x, public_key_y) = public_key
  compressed_prefix = "02" if (public_key_y % 2) == 0 else "03"
  return compressed_prefix + bitcoin.encode(public_key_x, 16, 64)
```

bitcoin_address.py

```
import bitcoin

def create_bitcoin_public_key(private_key):
  decoded_private_key =
    bitcoin.decode_privkey(private_key, "hex")
  public_key =
    bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
  (public_key_x, public_key_y) = public_key
  compressed_prefix = "02" if (public_key_y % 2) == 0 else "03"
  return compressed_prefix + bitcoin.encode(public_key_x, 16, 64)
```

bitcoin_address.py

```
import bitcoin

def create_bitcoin_public_key(private_key):
  decoded_private_key =
    bitcoin.decode_privkey(private_key, "hex")
  public_key =
    bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
  (public_key_x, public_key_y) = public_key
  compressed_prefix = "02" if (public_key_y % 2) == 0 else "03"
  return compressed_prefix + bitcoin.encode(public_key_x, 16, 64)
```

bitcoin_address.py

```
import bitcoin

def create_bitcoin_public_key(private_key):
  decoded_private_key =
    bitcoin.decode_privkey(private_key, "hex")
  public_key =
    bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
  (public_key_x, public_key_y) = public_key
  compressed_prefix = "02" if (public_key_y % 2) == 0 else "03"
  return compressed_prefix + bitcoin.encode(public_key_x, 16, 64)
```

bitcoin_address.py

```
import bitcoin

def create_bitcoin_public_key(private_key):
  decoded_private_key =
    bitcoin.decode_privkey(private_key, "hex")
  public_key =
    bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
  (public_key_x, public_key_y) = public_key
  compressed_prefix = "02" if (public_key_y % 2) == 0 else "03"
  return compressed_prefix + bitcoin.encode(public_key_x, 16, 64)
```

bitcoin_address.py

```
import bitcoin

def create_bitcoin_public_key(private_key):
  decoded_private_key =
    bitcoin.decode_privkey(private_key, "hex")
  public_key =
    bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
  (public_key_x, public_key_y) = public_key
  compressed_prefix = "02" if (public_key_y % 2) == 0 else "03"
  return compressed_prefix + bitcoin.encode(public_key_x, 16, 64)
```

Export

— https://github.com/fazibear/export

Erlport

— https://github.com/hdima/erlport


ERLANG

Install

```
defp deps do
  [
    # Erlport wrapper for Elixir to interface with Python code
    {:export, "~> 0.1.0"}
  ]
end
```

```elixir
defmodule BitcoinAddress.Python do
  use Export.Python

  @python_path :bitcoin_address |> :code.priv_dir() |> Path.basename()
  @python_file "bitcoin_address"

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <- create_bitcoin_public_key(pid, private_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Public key: #{inspect(bitcoin_public_key)}")
      Python.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, private_key) do
    pid
    |> Python.call(@python_file, "create_bitcoin_public_key", [private_key])
    |> to_string()
  end
end
```

```elixir
defmodule BitcoinAddress.Python do
  use Export.Python

  @python_path :bitcoin_address
               |> :code.priv_dir()
               |> Path.basename()
  @python_file "bitcoin_address"

  # ...
end
```

```elixir
defmodule BitcoinAddress.Python do
  use Export.Python

  @python_path :bitcoin_address
               |> :code.priv_dir()
               |> Path.basename()
  @python_file "bitcoin_address"

  # ...
end
```

```elixir
defmodule BitcoinAddress.Python do
  use Export.Python

  @python_path :bitcoin_address
               |> :code.priv_dir()
               |> Path.basename()
  @python_file "bitcoin_address"

  # ...
end
```

```
defmodule BitcoinAddress.Python do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      Python.stop(pid)
    end
  end

  # ...
end
```

```elixir
defmodule BitcoinAddress.Python do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      Python.stop(pid)
    end
  end

  # ...
end
```

```elixir
defmodule BitcoinAddress.Python do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      Python.stop(pid)
    end
  end

  # ...
end
```

```elixir
defmodule BitcoinAddress.Python do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      Python.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, priv_key) do
    pid
    |> Python.call(@python_file, "create_bitcoin_public_key", [priv_key])
    |> to_string()
  end
end
```

```elixir
defmodule BitcoinAddress.Python do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Public key: #{inspect(bitcoin_public_key)}")
      Python.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, priv_key) do
    # ...
  end
end
```

```elixir
defmodule BitcoinAddress.Python do
  use Export.Python

  @python_path :bitcoin_address |> :code.priv_dir() |> Path.basename()
  @python_file "bitcoin_address"

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <- create_bitcoin_public_key(pid, private_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Public key: #{inspect(bitcoin_public_key)}")
      Python.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, priv_key) do
    pid
    |> Python.call(@python_file, "create_bitcoin_public_key", [priv_key])
    |> to_string()
  end
end
```

Checklist

- Private Key ✅
- Bitcoin Public Key ✅
- Bitcoin Address

`bitcoin.pubkey_to_address`

```elixir
defmodule BitcoinAddress.Python do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key),
         bitcoin_address <-
           create_bitcoin_address(pid, bitcoin_public_key) do
      # ...
    end
  end

  defp create_bitcoin_address(pid, pub_key) do
    pid
    |> Python.call(@python_file, "bitcoin.pubkey_to_address", [pub_key])
    |> to_string()
  end
end
```

```elixir
defmodule BitcoinAddress.Python do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key),
         bitcoin_address <-
           create_bitcoin_address(pid, bitcoin_public_key) do
      # ...
    end
  end

  defp create_bitcoin_address(pid, pub_key) do
    pid
    |> Python.call(@python_file, "bitcoin.pubkey_to_address", [pub_key])
    |> to_string()
  end
end
```

```
defp create_bitcoin_public_key(pid, priv_key) do
  pid
  |> Python.call(@python_file, "create_bitcoin_public_key", [priv_key])
  |> to_string()
end
```

```
defp create_bitcoin_address(pid, pub_key) do
  pid
  |> Python.call(@python_file, "bitcoin.pubkey_to_address", [pub_key])
  |> to_string()
end
```

```
defp create_bitcoin_public_key(pid, priv_key) do
  pid
  |> Python.call(@python_file, "create_bitcoin_public_key", [priv_key])
  |> to_string()
end
```

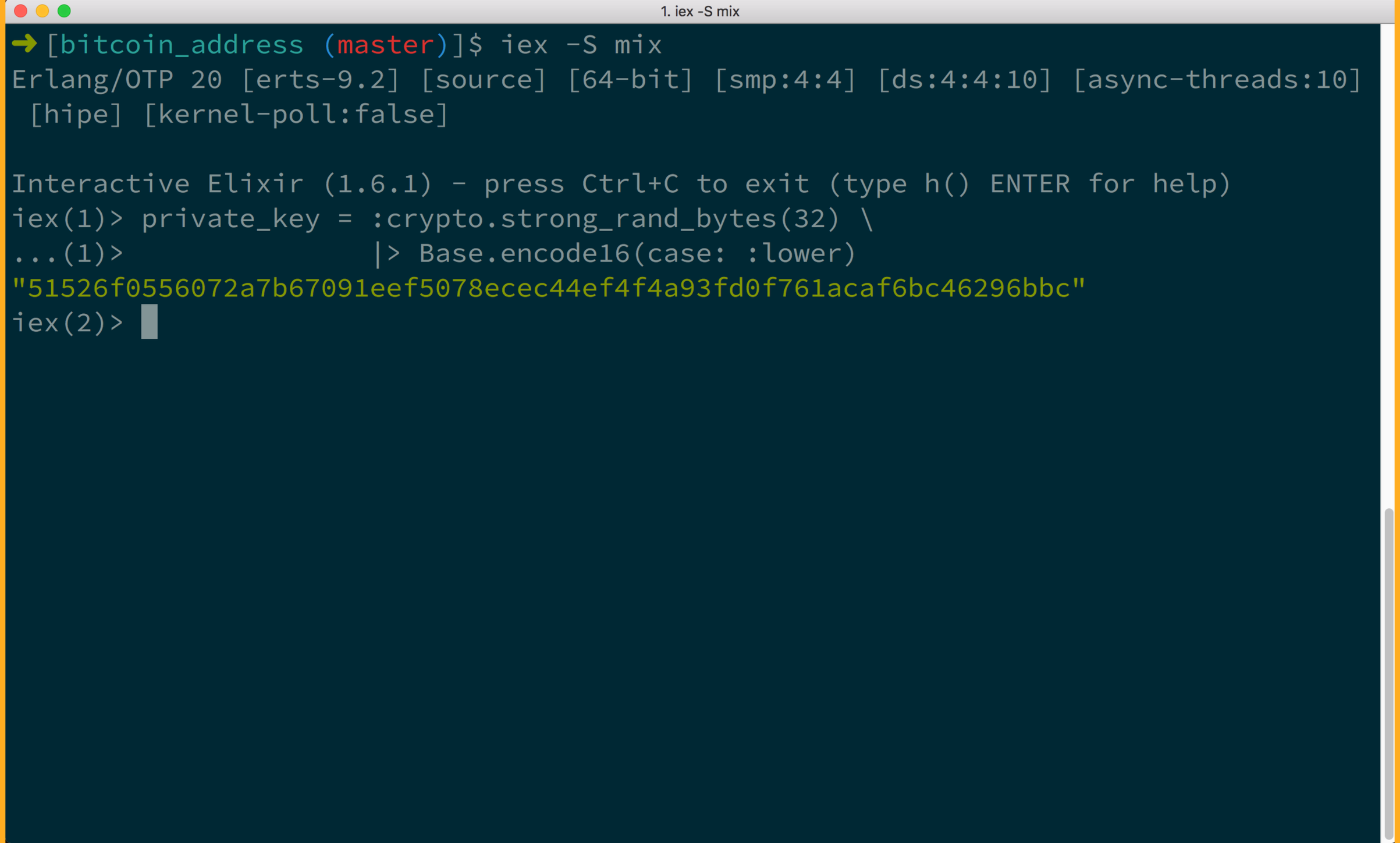
```
defp create_bitcoin_address(pid, pub_key) do
  pid
  |> Python.call(@python_file, "bitcoin.pubkey_to_address", [pub_key])
  |> to_string()
end
```

```elixir
defmodule BitcoinAddress.Python do
  # ...
  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key),
         bitcoin_address <-
           create_bitcoin_address(pid, bitcoin_public_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Public key: #{inspect(bitcoin_public_key)}")
      IO.puts("Bitcoin address: #{inspect(bitcoin_address)}")
      Python.stop(pid)
    end
  end

  defp create_bitcoin_address(pid, pub_key) do
    pid
    |> Python.call(@python_file, "bitcoin.pubkey_to_address", [pub_key])
    |> to_string()
  end
end
```

```elixir
defmodule BitcoinAddress.Python do
  use Export.Python

  @python_path :bitcoin_address |> :code.priv_dir() |> Path.basename()
  @python_file "bitcoin_address"

  def generate(private_key) do
    with {:ok, pid} <- Python.start(python_path: @python_path),
         bitcoin_public_key <- create_bitcoin_public_key(pid, private_key),
         bitcoin_address <- create_bitcoin_address(pid, bitcoin_public_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Public key: #{inspect(bitcoin_public_key)}")
      IO.puts("Bitcoin address: #{inspect(bitcoin_address)}")
      Python.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, priv_key) do
    call_python(pid, "create_bitcoin_public_key", [priv_key])
  end

  defp create_bitcoin_address(pid, pub_key) do
    call_python(pid, "bitcoin.pubkey_to_address", [pub_key])
  end

  defp call_python(pid, function, args) do
    pid
    |> Python.call(@python_file, function, args)
    |> to_string()
  end
end
```

Checklist

- Private Key ✅
- Bitcoin Public Key ✅
- Bitcoin Address ✅


1. iex -S mix
➜ [bitcoin_address (master)]$ iex -S mix
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
[hipe] [kernel-poll:false]
Interactive Elixir (1.6.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> private_key = :crypto.strong_rand_bytes(32) \
...(1)> |> Base.encode16(case: :lower)
"51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
iex(2)>

```
➜ [bitcoin_address (master)]$ iex -S mix
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
 [hipe] [kernel-poll:false]

Interactive Elixir (1.6.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> private_key = :crypto.strong_rand_bytes(32) \
...(1)>               |> Base.encode16(case: :lower)
"51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
iex(2)> BitcoinAddress.Python.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(3)>
```

What I think I look like explaining crypto VS what I actually look like

```
➜ [bitcoin_address (master)]$ iex -S mix
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
 [hipe] [kernel-poll:false]

Interactive Elixir (1.6.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> private_key = :crypto.strong_rand_bytes(32) \
...(1)>               |> Base.encode16(case: :lower)
"51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
iex(2)> BitcoinAddress.Python.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(3)>
```

Cure

— https://github.com/luc-tielen/Cure

$ brew install libbitcoin++

Install

```
defp deps do
  [
    # Interface C-code with Erlang/Elixir using Ports
    {:cure, "~> 0.4.0"},
  ]
end
```

```
$ mix cure.bootstrap
```

Cure Bootstrap

Cure Bootstrap

Cure Bootstrap

Cure Bootstrap

main.c

```c
#include "main.h"
int main(void) {
  int bytes_read;
  byte buffer[MAX_BUFFER_SIZE];

  while((bytes_read = read_msg(buffer)) > 0) {
    // TODO put C-code here, right now it only echos data back
    // to Elixir.
    send_msg(buffer, bytes_read);
  }
  return 0;
}
```

main.c

```c
#include "main.h"
int main(void) {
  int bytes_read;
  byte buffer[MAX_BUFFER_SIZE];

  while((bytes_read = read_msg(buffer)) > 0) {
    // TODO put C-code here, right now it only echos data back
    // to Elixir.
    send_msg(buffer, bytes_read);
  }
  return 0;
}
```

main.c

```c
#include "main.h"
int main(void) {
  int bytes_read;
  byte buffer[MAX_BUFFER_SIZE];

  while((bytes_read = read_msg(buffer)) > 0) {
    // TODO put C-code here, right now it only echos data back
    // to Elixir.
    send_msg(buffer, bytes_read);
  }
  return 0;
}
```

main.c

```c
#include "main.h"
int main(void) {
  int bytes_read;
  byte buffer[MAX_BUFFER_SIZE];

  while((bytes_read = read_msg(buffer)) > 0) {
    // TODO put C-code here, right now it only echos data back
    // to Elixir.
    send_msg(buffer, bytes_read);
  }
  return 0;
}
```

Rename Files

```
$ mv c_src/main.h c_src/bitcoin_address.h
$ mv c_src/main.c c_src/bitcoin_address.cpp
```

bitcoin_address.h

```
#ifndef BITCOIN_ADDRESS_H
#define BITCOIN_ADDRESS_H
#include <elixir_comm.h>

std::string create_bitcoin_public_key(std::string priv_key);

#endif
```

bitcoin_address.cpp

```
#include <string>
#include "bitcoin_address.h"

int main(void) { ... }

std::string create_bitcoin_public_key(std::string priv_key) {
  bc::ec_secret decoded;
  bc::decode_base16(decoded, priv_key);
  bc::wallet::ec_private
    secret(decoded, bc::wallet::ec_private::mainnet_p2kh);
  bc::wallet::ec_public public_key(secret);
  return public_key.encoded();
}
```

bitcoin_address.cpp

```
std::string create_bitcoin_public_key(std::string priv_key) {
  bc::ec_secret decoded;
  bc::decode_base16(decoded, priv_key);
  bc::wallet::ec_private
    secret(decoded, bc::wallet::ec_private::mainnet_p2kh);
  bc::wallet::ec_public public_key(secret);
  return public_key.encoded();
}
```

bitcoin_address.cpp

```
std::string create_bitcoin_public_key(std::string priv_key) {
  bc::ec_secret decoded;
  bc::decode_base16(decoded, priv_key);
  bc::wallet::ec_private
    secret(decoded, bc::wallet::ec_private::mainnet_p2kh);
  bc::wallet::ec_public public_key(secret);
  return public_key.encoded();
}
```

bitcoin_address.cpp

```
std::string create_bitcoin_public_key(std::string priv_key) {
  bc::ec_secret decoded;
  bc::decode_base16(decoded, priv_key);
  bc::wallet::ec_private
    secret(decoded, bc::wallet::ec_private::mainnet_p2kh);
  bc::wallet::ec_public public_key(secret);
  return public_key.encoded();
}
```

bitcoin_address.cpp

```
std::string create_bitcoin_public_key(std::string priv_key) {
  bc::ec_secret decoded;
  bc::decode_base16(decoded, priv_key);
  bc::wallet::ec_private
    secret(decoded, bc::wallet::ec_private::mainnet_p2kh);
  bc::wallet::ec_public public_key(secret);
  return public_key.encoded();
}
```

bitcoin_address.cpp

```
std::string create_bitcoin_public_key(std::string priv_key) {
  bc::ec_secret decoded;
  bc::decode_base16(decoded, priv_key);
  bc::wallet::ec_private
    secret(decoded, bc::wallet::ec_private::mainnet_p2kh);
  bc::wallet::ec_public public_key(secret);
  return public_key.encoded();
}
```

bitcoin_address.cpp

```
#include <string>
#include "bitcoin_address.h"

int main(void) { ... }

std::string create_bitcoin_public_key(std::string priv_key) {
  bc::ec_secret decoded;
  bc::decode_base16(decoded, priv_key);
  bc::wallet::ec_private
    secret(decoded, bc::wallet::ec_private::mainnet_p2kh);
  bc::wallet::ec_public public_key(secret);
  return public_key.encoded();
}
```

bitcoin_address.cpp

```
int main(void) {
  int bytes_read;
  byte buffer[MAX_BUFFER_SIZE];

  while((bytes_read = read_msg(buffer)) > 0) {
    // TODO put C-code here, right now it only echos data back
    // to Elixir.
    send_msg(buffer, bytes_read);
  }
  return 0;
}
```

Just send function name & private key?

Just send function name & private key?

Send Flag  !

Send Flag 🏁!

```
const int CREATE_BITCOIN_PUBLIC_KEY = 1;
```

bitcoin_address.cpp

```
const int CREATE_BITCOIN_PUBLIC_KEY = 1;

int main(void) {
  int bytes_read;
  byte buffer[MAX_BUFFER_SIZE];

  while((bytes_read = read_msg(buffer)) > 0) {
    process_command(buffer, bytes_read);
  }
  return 0;
}
```

bitcoin_address.cpp

```
const int CREATE_BITCOIN_PUBLIC_KEY = 1;

int main(void) {
  int bytes_read;
  byte buffer[MAX_BUFFER_SIZE];

  while((bytes_read = read_msg(buffer)) > 0) {
    process_command(buffer, bytes_read);
  }
  return 0;
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

bitcoin_address.cpp

```
void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  alias Cure.Server, as: Cure

  @cpp_executable "priv/bitcoin_address"
  # Integers representing C++ methods
  @create_bitcoin_public_key 1

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <- create_bitcoin_public_key(pid, private_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Bitcoin public key: #{inspect(bitcoin_public_key)}")
      :ok = Cure.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, private_key) do
    Cure.send_data(pid, <<@create_bitcoin_public_key, private_key::binary>>, :once)

    receive do
      {:cure_data, response} ->
        response
    end
  end
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  alias Cure.Server, as: Cure

  @cpp_executable "priv/bitcoin_address"
  # Integers representing C++ methods
  @create_bitcoin_public_key 1

  # ...
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  alias Cure.Server, as: Cure

  @cpp_executable "priv/bitcoin_address"
  # Integers representing C++ methods
  @create_bitcoin_public_key 1

  # ...
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  alias Cure.Server, as: Cure

  @cpp_executable "priv/bitcoin_address"
  # Integers representing C++ methods
  @create_bitcoin_public_key 1

  # ...
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      :ok = Cure.stop(pid)
    end
  end

  # ...
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      :ok = Cure.stop(pid)
    end
  end

  # ...
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      :ok = Cure.stop(pid)
    end
  end

  # ...
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  # ...

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key) do
      # ...
      :ok = Cure.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, priv_key) do
    Cure.send_data(pid, <<@create_bitcoin_public_key, priv_key::binary>>, :once)
    receive do
      {:cure_data, response} ->
        response
    end
  end
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  alias Cure.Server, as: Cure

  @cpp_executable "priv/bitcoin_address"
  # Integers representing C++ methods
  @create_bitcoin_public_key 1

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <- create_bitcoin_public_key(pid, private_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Bitcoin public key: #{inspect(bitcoin_public_key)}")
      :ok = Cure.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, private_key) do
    Cure.send_data(pid, <<@create_bitcoin_public_key, priv_key::binary>>, :once)

    receive do
      {:cure_data, response} ->
        response
    end
  end
end
```

Checklist

- Private Key ✅
- Bitcoin Public Key ✅
- Bitcoin Address

bitcoin_address.cpp

```
std::string create_bitcoin_address(std::string pub_key) {
  bc::wallet::ec_public public_key = bc::wallet::ec_public::ec_public(pub_key);
  bc::data_chunk public_key_data;
  public_key.to_data(public_key_data);
  const auto hash = bc::bitcoin_short_hash(public_key_data);
  bc::data_chunk unencoded_address;
  unencoded_address.reserve(25);
  unencoded_address.push_back(0);
  bc::extend_data(unencoded_address, hash);
  bc::append_checksum(unencoded_address);
  const std::string address = bc::encode_base58(unencoded_address);
  return address;
}
```

```
const int CREATE_BITCOIN_PUBLIC_KEY = 1;
const int CREATE_BITCOIN_ADDRESS = 2;

int main(void) { ... }

void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
    case CREATE_BITCOIN_ADDRESS:
      result = create_bitcoin_address(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

```
const int CREATE_BITCOIN_PUBLIC_KEY = 1;
const int CREATE_BITCOIN_ADDRESS = 2;

int main(void) { ... }

void process_command(byte* buffer, int bytes_read) {
  int function = buffer[0];
  std::string arg = (char*) &buffer[1];
  std::string result;

  switch (function) {
    case CREATE_BITCOIN_PUBLIC_KEY:
      result = create_bitcoin_public_key(arg);
      break;
    case CREATE_BITCOIN_ADDRESS:
      result = create_bitcoin_address(arg);
      break;
  }
  memcpy(buffer, result.data(), result.length());
  send_msg(buffer, result.size());
}
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  @create_bitcoin_public_key 1
  @create_bitcoin_address 2

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key),
         bitcoin_address <-
           create_bitcoin_address(pid, public_key) do
      # ...
    end
  end

  defp create_bitcoin_address(pid, pub_key) do
    Cure.send_data(pid, <<@create_bitcoin_address, pub_key::binary>>, :once)
    receive do
      {:cure_data, response} ->
        response
    end
  end
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  @create_bitcoin_public_key 1
  @create_bitcoin_address 2

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key),
         bitcoin_address <-
           create_bitcoin_address(pid, public_key) do
      # ...
    end
  end

  defp create_bitcoin_address(pid, pub_key) do
    Cure.send_data(pid, <<@create_bitcoin_address, pub_key::binary>>, :once)
    receive do
      {:cure_data, response} ->
        response
    end
  end
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  @create_bitcoin_public_key 1
  @create_bitcoin_address 2

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key),
         bitcoin_address <-
           create_bitcoin_address(pid, public_key) do
      # ...
    end
  end

  defp create_bitcoin_address(pid, pub_key) do
    Cure.send_data(pid, <<@create_bitcoin_address, pub_key::binary>>, :once)
    receive do
      {:cure_data, response} ->
        response
    end
  end
end
```

```elixir
defmodule BitcoinAddress.CPlusPlus do
  alias Cure.Server, as: Cure

  @cpp_executable "priv/bitcoin_address"
  # Integers representing C++ methods
  @create_bitcoin_public_key 1
  @create_bitcoin_address 2

  def generate(private_key) do
    with {:ok, pid} <- Cure.start_link(@cpp_executable),
         bitcoin_public_key <-
           create_bitcoin_public_key(pid, private_key),
         bitcoin_address <-
           create_bitcoin_address(pid, bitcoin_public_key) do
      IO.puts("Private key: #{inspect(private_key)}")
      IO.puts("Public key: #{inspect(bitcoin_public_key)}")
      IO.puts("Bitcoin address: #{inspect(bitcoin_address)}")
      :ok = Cure.stop(pid)
    end
  end

  defp create_bitcoin_public_key(pid, priv_key) do
    call_cpp(pid, <<@create_bitcoin_public_key, priv_key::binary>>)
  end

  defp create_bitcoin_address(pid, pub_key) do
    call_cpp(pid, <<@create_bitcoin_address, pub_key::binary>>)
  end

  defp call_cpp(pid, data) do
    Cure.send_data(pid, data, :once)
    receive do
      {:cure_data, response} ->
        response
    end
  end
end
```

Checklist

- Private Key ✅
- Bitcoin Public Key ✅
- Bitcoin Address ✅


ut

Makefile

Makefile 😱

```
CC = g++ -std=c++11
APP_DIR = $(shell dirname $(shell pwd))
CURE_DEPS_DIR = $(APP_DIR)/deps/cure/c_src
CURE_DEPS = -I$(CURE_DEPS_DIR) -L$(CURE_DEPS_DIR)
ELIXIR_COMM_C = -x c++ $(CURE_DEPS_DIR)/elixir_comm.c
LIBBITCOIN_DEPS = $(shell pkg-config --cflags --libs libbitcoin)
C_FLAGS = $(CURE_DEPS) $(ELIXIR_COMM_C) $(LIBBITCOIN_DEPS) -O3
PRIV_DIR = $(APP_DIR)/priv
C_SRC_DIR = $(APP_DIR)/c_src
EXECUTABLES = bitcoin_address

all: $(EXECUTABLES)
# * $< - prerequisite file
# * $@ - executable file
$(EXECUTABLES): %: %.cpp
    $(CC) $(C_FLAGS) $(C_SRC_DIR)/$< -o $(PRIV_DIR)/$@
```

Compile from c_src/ to priv/

```
all: $(EXECUTABLES)
# * $< - prerequisite file
# * $@ - executable file
$(EXECUTABLES): %: %.cpp
	$(CC) $(C_FLAGS) $(C_SRC_DIR)/$< -o $(PRIV_DIR)/$@
```

Compile C++ with Elixir

```
defmodule BitcoinAddress.Mixfile do
  use Mix.Project

  def project do
    [
      # ...
      compilers: Mix.compilers ++ [:cure, :"cure.deps"]
    ]
  end

  # ...
end
```

Compile during TDD

```
if Mix.env() == :dev do
  config :mix_test_watch,
    clear: true,
    tasks: [
      "compile.cure",
      "format",
      "test",
      "credo --strict"
    ]
end
```

Anyway...

```
➜ [bitcoin_address (master)]$ iex -S mix
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
 [hipe] [kernel-poll:false]

Interactive Elixir (1.6.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> private_key = :crypto.strong_rand_bytes(32) \
...(1)>               |> Base.encode16(case: :lower)
"51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
iex(2)> BitcoinAddress.Python.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(3)>
```

```
➜ [bitcoin_address (master)]$ iex -S mix
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
 [hipe] [kernel-poll:false]

Interactive Elixir (1.6.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> private_key = :crypto.strong_rand_bytes(32) \
...(1)>               |> Base.encode16(case: :lower)
"51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
iex(2)> BitcoinAddress.Python.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(3)> BitcoinAddress.CPlusPlus.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(4)>
```

Self-Promotion

— https://paulfioravanti.com/blog/2017/12/04/using-pythons-bitcoin-libraries-in-elixir/

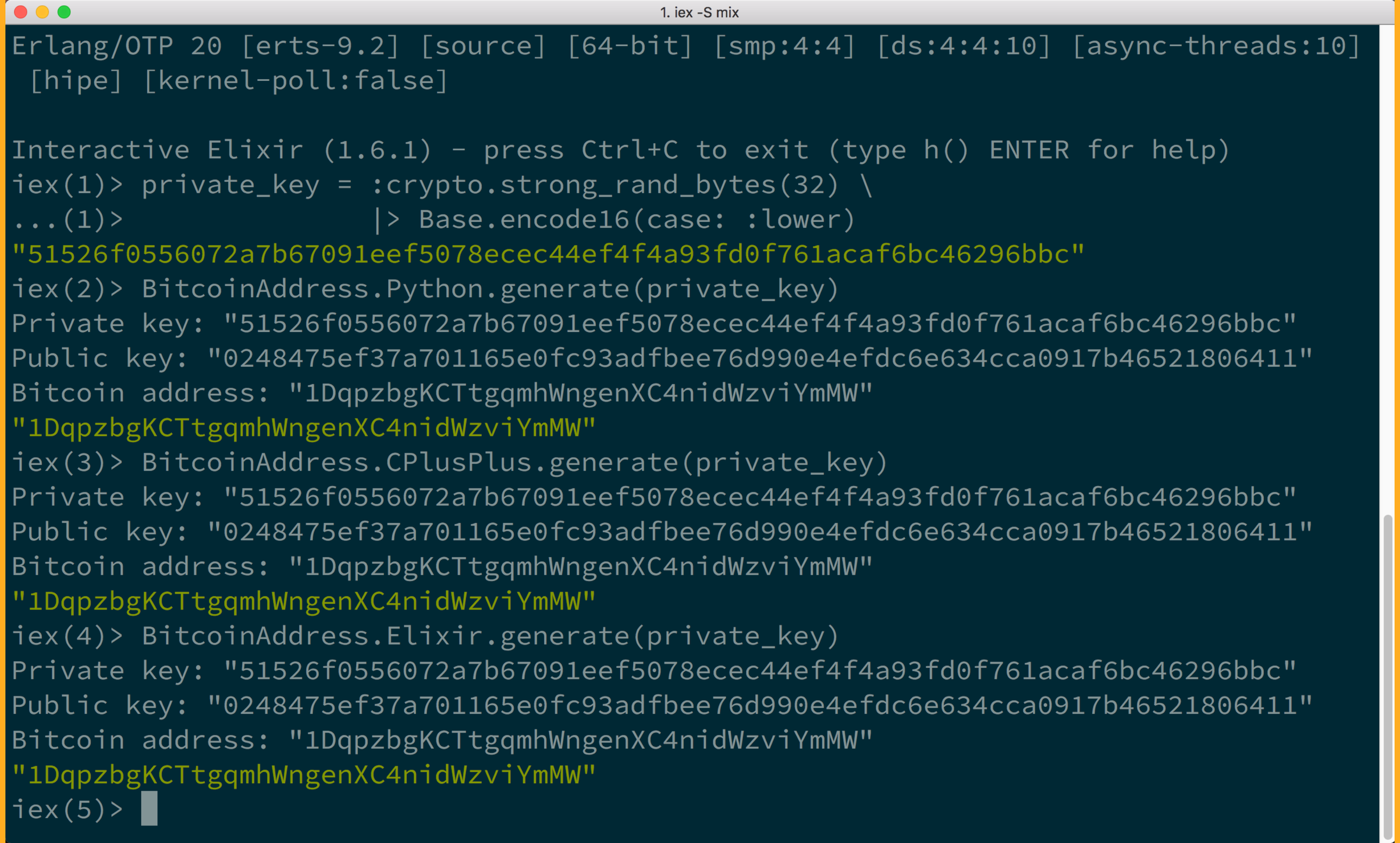
— https://paulfioravanti.com/blog/2017/12/13/using-c-plus-plus-bitcoin-libraries-in-elixir/

Github Repos

— https://github.com/paulfioravanti/bitcoin_address

— https://github.com/paulfioravanti/mastering_bitcoin

Ironically enough...


1. iex -S mix
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
[hipe] [kernel-poll:false]
Interactive Elixir (1.6.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> private_key = :crypto.strong_rand_bytes(32) \
...(1)> |> Base.encode16(case: :lower)
"51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
iex(2)> BitcoinAddress.Python.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(3)> BitcoinAddress.CPlusPlus.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(4)> BitcoinAddress.Elixir.generate(private_key)
Private key: "51526f0556072a7b67091eef5078ecec44ef4f4a93fd0f761acaf6bc46296bbc"
Public key: "0248475ef37a701165e0fc93adfbee76d990e4efdc6e634cca0917b46521806411"
Bitcoin address: "1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
"1DqpzbgKCTtgqmhWngenXC4nidWzviYmMW"
iex(5)>

Thanks!

@paulfioravanti

1FRvnTotNzKA4JLTddVE5KuSVy8oH2Fq4o