

Compiling Policies with rust-miniscript

October 20, 2026 · Document #403 · bip39-pharse.com



Photo by [T](#) on [Unsplash](#)

Learn how to define spending conditions in rust-miniscript's policy language, compile to descriptors, and inspect execution costs.

Contents

Defining the Spending Policy	1
Compiling Policy to Miniscript	2
Inspecting Descriptors and Execution Weight	3
FAQ	4
Sources	4
Example BIP39 word sequences	4
Reading a BIP39 phrase	5
See also	6

Bitcoin Script lacks static analysis tools to verify whether an arbitrary script allows unintended spend paths, suffers from malleability, or exceeds consensus execution limits. The `rust-miniscript` library solves this by introducing a high-level policy language that compiles down to Miniscript, which then maps directly to valid, verifiable Bitcoin Script embedded inside output descriptors like `wsh()` or `tr()`.

Defining the Spending Policy

The policy language represents monotonic spending conditions using intuitive primitives. Rather than tracking execution stacks, alt-stacks, and opcodes manually, spending conditions are expressed as boolean trees containing keys, timelocks, and thresholds.

Consider an inheritance or backup scheme where key A can spend immediately, but key B can only spend after an inactivity period of 4,320 blocks (approximately 30 days):

```
or(pk(key_a), and(pk(key_b), older(4320)))
```

In rust-miniscript, policies are represented via the `miniscript::policy::Concrete` type. The policy parser reads strings containing public keys or string aliases. Aliases are commonly used during compilation, after which concrete public keys or BIP32 extended public keys (xpub) are substituted into the resulting descriptor.

Compiling Policy to Miniscript

The compiler translates the abstract policy tree into an optimized Miniscript tree. Miniscript assigns type properties to every sub-expression to guarantee type correctness, safety, and non-malleability across spending paths.

Step 1: Parse the Abstract Policy

First, load the policy string into the concrete policy structure using Rust:

- 1 Add `miniscript` and `bitcoin` to `Cargo.toml`.
- 2 Parse the policy using `Concrete::<String>::from_str`.

```
```rust
use std::str::FromStr;

use miniscript::policy::Concrete;

use miniscript::Segwitv0;

let policy_str = "or(pk(A), and(pk(B), older(4320)))";
let policy = Concrete::<String>::from_str(policy_str)
 .expect("Valid policy string");
```
```

Step 2: Run the Compiler for Segregated Witness (v0)

To target SegWit version 0 (wsh), run the compiler targeting the `Segwitv0` context. The compiler searches the space of valid Miniscript expressions to find the target that minimizes witness and script size while avoiding malleability:

```
```rust
let miniscript = policy.compile::<Segwitv0>()
 .expect("Compilation failed");
```
```

For `or(pk(A), and(pk(B), older(4320)))`, the compiler produces a valid Miniscript expression resembling:

```
or_d(pk(A), and_v(vc:pk(B), older(4320)))
```

This expression translates into the following Bitcoin opcodes inside the witness script:

```
<A> CHECKSIG IFDUP NOTIF <B> CHECKSIGVERIFY <4320> CHECKSEQUENCEVERIFY ENDIF
```

If key A provides a valid signature, CHECKSIG returns 1, IFDUP duplicates it, and NOTIF skips the rest of the execution. If key A fails, execution drops through to enforce the relative locktime via CHECKSEQUENCEVERIFY and check key B's signature.

Step 3: Compiling for Taproot (v1)

In Taproot (`tr()`), the policy compiler leverages TapTrees rather than creating one massive script. Under Taproot, disjunctions (`or`) can be split into distinct leaf scripts within the Merkle tree. A user spending via key A does not need to reveal key B or the relative timelock, preserving privacy and saving transaction fees.

When targeting Tap, the policy compiler translates alternative paths directly into TapTree leaves:

- Leaf 1: `pk(A)`
- Leaf 2: `and_v(vc:pk(B), older(4320))`

Inspecting Descriptors and Execution Weight

Once the Miniscript is compiled, wrapping it into a descriptor allows precise calculation of script size, address derivation, and maximum satisfaction weight.

Replace the string aliases A and B with valid public keys (such as compressed SECP256k1 public keys or BIP32 derivation paths), then instantiate the descriptor:

```
```rust
use miniscript::Descriptor;

let desc_str = format!("wsh({})", miniscript);

let descriptor = Descriptor::<String>::from_str(&desc_str)

.expect("Valid descriptor");
```
```

Metrics Inspection

The `Descriptor` type exposes methods to inspect size and weight metrics:

- `descriptor.script_size()`: Returns the byte size of the witnessScript or leaf script.
- `descriptor.max_weight_to_satisfy()`: Computes the worst-case witness weight required to satisfy the condition, measuring the maximum fee required to spend from the UTXO.

| Context | Script Structure | Witness Script Size | Worst-Case Weight | Spending Privacy |
|------------------------------|--------------------|---------------------|-------------------|----------------------|
| <code>wsh()</code> | Single flat script | ~75 bytes | ~324 weight units | All paths exposed |
| <code>tr()</code> Leaf Spend | Merkle leaf tree | ~35 bytes (Leaf 2) | ~165 weight units | Unused leaves hidden |

In the `wsh()` implementation, the worst-case spending path occurs when executing the secondary path (key `B` and relative timelock), which requires pushing a signature for `B`, fulfilling the timelock, a zero-byte push to fail the initial branch condition, and the full witness script.

In `tr()`, each leaf is satisfied independently without overhead from alternative branches, requiring only the leaf script, its arguments, and the Merkle inclusion proof.

FAQ

Why does the compiler produce `or_d` instead of `or_b` or `or_c`?

Miniscript uses type letters (`B`, `v`, `k`, `w`) to denote expression behavior on the stack. The compiler chooses `or_d` because it converts a nonzero outcome on the first branch into a clean true value without running the second branch, minimizing script weight and execution steps.

Can any arbitrary policy be compiled?

No. Policies with conflicting timelock types (mixing height-based `older` with time-based `older`) or non-monotonic spending logic (requiring an operation to evaluate to false to spend) will fail compilation because Bitcoin Script does not permit non-monotonic conditions without introducing malleability vulnerabilities.

How does rust-miniscript prevent third-party malleability?

The compiler verifies that every execution path requires signatures on all non-hashlock inputs, eliminates ambiguous truthy values by enforcing strict `1` or `0` evaluations, and disallows spending combinations where extra witness items could be injected without invalidating the transaction.

Sources

- [Miniscript: Streamlined Bitcoin Scripting](https://bitcoin.sipa.be/) (bitcoin.sipa.be)
- [rust-miniscript Documentation](https://docs.rs/) (docs.rs)

Continue reading on bip39-phrase.com: [BIP39 vs BIP32 vs BIP44: How the Seed Standards Fit Together](#)

More on this topic: [BIP Standards](#)

Example BIP39 word sequences

Randomly generated sequences of words from the BIP39 English wordlist. None of them passes the BIP39 checksum, so no wallet will accept them: they show the format only. Never store funds with a phrase you found in a document.

| Example 1 | | | |
|-----------|------------|-----------|------------|
| 1 more | 2 spirit | 3 rapid | 4 security |
| 5 rude | 6 garage | 7 balcony | 8 style |
| 9 erupt | 10 digital | 11 adjust | 12 ribbon |

| Example 2 | | | |
|------------|----------|-----------|----------|
| 1 kangaroo | 2 wire | 3 outdoor | 4 noodle |
| 5 document | 6 chaos | 7 nut | 8 clog |
| 9 someone | 10 crack | 11 diary | 12 rose |

| Example 3 | | | |
|-----------|------------|------------|-----------|
| 1 dress | 2 property | 3 tomorrow | 4 giggle |
| 5 start | 6 usage | 7 swamp | 8 tattoo |
| 9 spoon | 10 sea | 11 million | 12 absurd |

| Example 4 | | | |
|------------|-----------|-----------|-----------|
| 1 harsh | 2 tornado | 3 media | 4 used |
| 5 document | 6 royal | 7 assault | 8 hover |
| 9 rail | 10 metal | 11 nasty | 12 please |

| Example 5 | | | |
|-----------|-------------|-------------|-----------|
| 1 trial | 2 hip | 3 myth | 4 job |
| 5 sell | 6 away | 7 rifle | 8 earth |
| 9 eight | 10 cupboard | 11 umbrella | 12 affair |

| Example 6 | | | |
|------------|----------|-----------|------------|
| 1 thought | 2 person | 3 patrol | 4 morning |
| 5 borrow | 6 say | 7 length | 8 warm |
| 9 resemble | 10 long | 11 broken | 12 process |

| Example 7 | | | |
|-----------|------------|-----------|----------|
| 1 embark | 2 sand | 3 choice | 4 crazy |
| 5 arrive | 6 neither | 7 warm | 8 celery |
| 9 scrub | 10 stadium | 11 income | 12 bomb |

| Example 8 | | | |
|-----------|-----------|-----------|------------|
| 1 accuse | 2 defense | 3 bracket | 4 mechanic |
| 5 sauce | 6 regret | 7 peanut | 8 slam |
| 9 pluck | 10 false | 11 snake | 12 early |

Reading a BIP39 phrase

Every word stands for an 11-bit number: its position in the 2048-word list, counted from 0. Here is Example 1 decoded word by word. Each word links to its page on bip39-phrases.com, where the list is numbered from 1.

| # | Word | Index (0-2047) | Word no. | 11 bits |
|---|----------------------|----------------|----------|--------------|
| 1 | more | 1150 | 1151 | 100011111110 |

| # | Word | Index (0-2047) | Word no. | 11 bits |
|----|----------|----------------|----------|-------------|
| 2 | spirit | 1680 | 1681 | 11010010000 |
| 3 | rapid | 1423 | 1424 | 10110001111 |
| 4 | security | 1558 | 1559 | 11000010110 |
| 5 | rude | 1511 | 1512 | 10111100111 |
| 6 | garage | 763 | 764 | 01011111011 |
| 7 | balcony | 142 | 143 | 00010001110 |
| 8 | style | 1726 | 1727 | 11010111110 |
| 9 | erupt | 615 | 616 | 01001100111 |
| 10 | digital | 495 | 496 | 00111101111 |
| 11 | adjust | 28 | 29 | 00000011100 |
| 12 | ribbon | 1480 | 1481 | 10111001000 |

The 12 words carry 132 bits: 128 bits of entropy followed by a 4-bit checksum. The checksum must equal the first 4 bits of the SHA-256 hash of the entropy.

Entropy (hex): 8fda42c7e16bcebec476be4ce7bc0e5c

This sequence ends in the checksum bits 1000, but SHA-256 of its entropy starts with 1111, so wallets reject it as an invalid phrase. That is the point of the checksum: about 15 of every 16 random or mistyped sequences fail it.

See also

- [Why There Is No BIP39 Version 2](#) · October 16, 2026

Educational document. The example sequences are random and fail the BIP39 checksum: they are not wallets and must never hold funds. Security notes: [bip39-phrase.com/security/](#)