TapScript Compiler

> *Assuming* basic knowledge of Taproot, TapTree, TapScript and high-level overview of Miniscript.

Given an arbitrary policy p, our goal is to compile it to output a Taproot Descriptor which is:

1. *Cost-Effective* where this *cost* is defined later-on in the document.
2. *Private*, where we try to reveal as little information (by obscuring the need for revealing scripts by seperating them in *TapLeaves*) while keeping our compilation sound.

Taproot Output Structure and how it's handled for Miniscript

To spend a Taproot Output, either satisfy the *internal_key* or a valid *script-path spend*.

Internal Key Spend

Assuming knowledge of *internal_key* in Taproot outputs, we extract the most-probable public key from the policy which can single-handedly spend all the funds. Otherwise, an *unspendable key* (which can't be satisfied) is set.

Script-Path Spend

A **script path spend** in a TapTree implies we choose a single leaf-script to satisfy. This gives us the idea to construct a disjunctive form over a given policy (thanks to the policy language grammar), leaf nodes of which serve as the building-blocks of the constructed TapTree.

Private Compilation

Root-level disjunctive enumeration of the given policy p over $or()$ and $thresh(1,\ \ldots)$ and compilation of the resulting list of (sub-policies → respective miniscript compilation) into the TapTree by Huffman encoding over probabilities.

> **Upcoming**: Root-level disjunctive enumeration strategies for $thresh(k, \ldots)$.

Efficient Compilation

We are to construct the ~~best~~ cost-efficient TapTree compilation for our given policy. Owing to the exponential complexity of constructing every possible TapTree from the list of miniscript compilations, we resort to using heuristics.

Huffman Encoding

> *Heuristic*: Change the **merge** part of Huffman Algorithm. During merge of intermediate-*TapTrees* (say A and B *TapTree/Leaf*s) in Huffman Encoding Algorithm, consider optimal among both:
>
> 1. `TapTree(A, B)`
> 2. `TapLeaf(compilation!(or(policy_A,policy_B)))`

Def. TapTree Cost is the expected average-satisfaction cost for a given TapTree.

> $\text{TapLeaf Cost}(T)$ = $p_T \times (s_T + 33 + 32h_T + c_T)$.

Claim. Constructing the `TapTree` with A and B as children nodes $(1.)$ is **more cost-efficient** than $(2.)$.

Proof. Consider *TapLeaves* A and B, and let their *parent* compilation N (as defined by $(2.)$). Let s_A, s_B, s_N be corresponding script costs for all leaf-scripts in respective trees, h_A, h_B, h_N be height of sub-trees A and B, p_A, p_B, p_N are the respective probabilities ($p_N = p_A + p_B$ by construction) and c_A, c_B, c_N be their average-satisfaction costs. We have

1. $h_N = max(h_A, h_B) + 1 \implies h_N > h_A, h_B$.
   Since height of the parent tree is one more than the the maximum height of either children trees.
2. $$\begin{aligned} c_N &:= E[\text{Satisfaction cost of miniscript in leaf node } N] \\ &\geq E[\text{Satisfaction cost for child node} + C_{A/B}] \\ &\geq E[\text{Satisfaction cost for child node}] \\ &= \frac{p_A}{p_N} c_A + \frac{p_B}{p_N} c_B \\ &\implies p_N c_N \geq p_A c_A + p_B c_B \end{aligned} \tag{2}$$

where $C_{A/B}$ is the extra cost incurred for choosing which node to satisfy in the compiled miniscript `or_{i,b,c,d}(A,B)` decoded to bitcoin script and the probabilities p_A, p_B are normalized in the last step because the probabilities correspond according to the odds in the `or_{i,b,c,d}` fragment.

3. $s_N \geq s_A + s_B$.
   The script size for the parent compilation is greater than sum of respective children as it is evident from the bitcoin script decoding of `or_{i,b,c,d}` fragments (extra OPCODES).

These gives us:

$$\begin{aligned}
& p_N s_N > (p_A + p_B)(s_A + s_B) \\
& \implies p_A s_A + p_B s_B - p_N s_N < -p_A s_B - p_B s_A && (4) \\
& p_N h_N = (p_A + p_B) * \ max(h_A, h_B) + 1 \\
& = p_A \, max(h_A, h_B) + p_B \, max(h_A, h_B) + p_N \\
& > p_A h_A + p_B h_B + p_N \\
& \implies p_A h_A + p_B h_B - p_N h_N \leq p_N && (5)
\end{aligned}$$

$$\text{TapLeaf cost}(A) + \text{TapLeaf cost}(B) - \text{TapLeaf cost}(N)$$

$$\begin{aligned}
& = (p_A s_A + p_B s_B - p_N s_N) + 32 \\
& \times (p_A h_A + p_B h_B - p_N h_N) + (p_A c_A + p_B c_B - p_N c_N) \\
& \leq 32 \times (p_A h_A + p_B h_B - p_N h_N) \\
& + (p_A s_A + p_B s_B - p_N s_N) && (\text{from } (2)) \\
& \leq 32 \times p_N + (p_A s_A + p_B s_B - p_N s_N) && (\text{from } (5)) \\
& \leq 32 \times p_N - p_A s_B - p_B s_A && (\text{from } (4))
\end{aligned}$$

$$\begin{aligned}
& \textbf{Case 1. } s_A \geq 32, s_B \geq 32 \\
& \implies 32 p_N - p_A s_B - p_B s_A \leq 32(p_A + p_B) - 32 p_A - 32 p_B \leq 0 \\
& \implies \text{Tapleaf cost}(N) \geq \text{Tapleaf cost}(A) + \text{Tapleaf cost}(B)
\end{aligned}$$

This case happens with all the valid miniscripts containing atleast the 33-byte *PublicKey*.

The *valid* miniscripts with **script-size** less than 32 must contain only `pk_h`, but intuitively we can see that the satisfaction for this case must contain the key as well as hash which seems more inefficient.

Consider the two leaf script compilations $A := pk(\text{PublicKey})$ and $B := pkh(\text{PublicKeyHash})$(both having same probabilities $p_A = p_B$). For the policy $or(pol_C, pk(\text{PublicKey}))$ we have three possible choices to TapTree compilation (generally):

1. *TapLeaf*($or_{i/b/c/d}(ms_C, pk(\text{PublicKey}))$)
2. **TapTree**(*Leaf*(ms_C), *Leaf*($pk(\text{PublicKey})$))
3. **TapTree**(*Leaf*(ms_C), *Leaf*($pkh(\text{PublicKeyHash})$))

From case (1) ($s_{ms_C} \geq 32, s_A \geq 32$), $(2.)$ is *more efficient* than $(1.)$. Besides this, considering Schnorr signatures and byte size after serialization respectively,

$$
\begin{aligned}
&\text{TapTree cost}(3) - \text{TapTree cost}(2) \\
&= \text{TapLeaf cost}(3)_{ms_C} + \text{TapLeaf cost}(3)_{pk} \\
&- \text{TapLeaf cost}(3)_{ms_C} - \text{TapLeaf cost}(3)_{pkh} \\
&= p_{pkh} \times (c_{pkh} + 32 * h_{pkh} + s_{pkh}) - p_{pk} \times (c_{pk} + 32 * h_{pk} + s_{pk}) \\
&= p_B \times (c_B + 32 * h_B + s_B) - p_A \times (c_A + 32 * h_A + s_A) \\
&= p_A \times (c_B - c_A + s_B - s_A) \\
&= p_A \times (\text{Secret Key}_{sz} + \text{PublicKey}_{sz} - \text{PublicKey}_{sz} \\
&+ \text{PublicKeyHash}_{sz} + \mathit{OP\_CODES}_{pkh} - \text{PublicKey}_{sz}) \\
&= p_A \times (66 + 33 - 33 + 20 + 8 - 33) > 0 \\
&\implies \text{TapTree cost}(3) > \text{TapTree cost}(2)
\end{aligned}
$$

where the size of *OP_CODES* are considered according to `c:pk_h` bitcoin script serialization.

Thus, we can say $(2.)$ is **more efficient** than $(3.)$, and that it is always more *cost-efficient* to seperate/ enumerate the policy into different *TapLeaves*.
Hence, we can safely say that the **private** compilation is also indeed **cost-efficient**.