Chapter 2

Frankencoin

JEL Classification Codes: D40, D44, E42, G23, G32

Keywords: digital currencies, stablecoins, auctions, risk management, decentralized systems

Co-authors: Basile Maire (primarily section 2.7), Luzius Meisser

Publication (2022 version): Maire B., Meisser, L. Frankencoin. *SNB-CIF Conference on Cryptoassets and Financial Innovation*, June 3rd 2022

Abstract

Frankencoin is a decentralized, collateralized stablecoin with a modular architecture. Core contributions are a novel auction-based liquidation mechanism and a mostly interaction-free governance protocol. These two in combination make the Frankencoin independent of external oracles and allow for a high degree of flexibility with regard to the assets used as collateral. Additional equity capital to secure the system is supplied by offering a governance token through the pricing mechanism presented in chapter 3, effectively making the Frankencoin system a fully decentralized *Continuous Capital Corporation.* The long-term fundamental value that mirrors that of the reference currency is obtained via collateralization and trust in the stakeholders to align the return of holding Frankencoins with that of holding Swiss francs. Market forces, rather than an oracle-based conversion mechanism, are hoped to maintain the short-term peg. Finally, we assess the system's risks through the lens of a bank regulator, using historical data. The richly commented and audited Frankencoin

source code is provided as an appendix to complete the specification. The name “Frankencoin" hints at the contemplated reference currency, the Swiss franc, as well as the system’s self-governing nature.

2.1 Introduction

We start from the use-case of a Swiss franc Lombard loan, collateralized with tokens such as crypto-currencies or tokenized securities. The borrower deposits collateral into the system and thereby gains the possibility to mint a token, the “Frankencoin” ZCHF, that is pegged to the Swiss franc. If the value of the borrower’s collateral falls below a certain threshold, the loan can be liquidated and the borrower incurs a haircut. If the value of the collateral falls below the loan value before the liquidation ends, the loss eats into the system’s capital reserve. Implementing this system in a fully decentralized way on the blockchain leads us to our main contributions:

– A modular approach that allows anyone to propose new minting mechanisms and new collateral assets under a novel governance mechanism

– An innovative auction mechanism to enable oracle-free collateralized loans

– Capital requirements inspired by traditional banking rules

In comparison to other decentralized stablecoins, the Frankencoin stands out with its simplicity and versatility, which is achieved by having a basic but extensible setup that relies on the overarching economic incentives of the system participants instead of strictly enforcing a narrow peg with technical means.

2.1.1 Existing Systems

This section describes existing stablecoins and how they solve the challenge of having a stable value compared to a reference currency. In comparison to the Frankencoin, most stablecoins are more directly concerned about steering the short-term peg to the reference currency at small deviations, whereas the Frankencoin aims at creating a credible fundamental value in the long term and relies more on market forces to keep the exchange rate close to the fundamental value in the short term.

Stabilization Methods

The largest two stablecoins, the Tether (USDT) and the USD Coin (USDC) are based on the promise of an issuer to always sell and repurchase them at the price of the reference currency, thereby enforcing a 1:1 peg for as long as the issuer is solvent.[1] A similar stablecoin tracking the Swiss franc is the CryptoFranc (XCHF).[2] In the background, the issuer can apply any stabilization mechanism the law allows. While capital requirements for banks are prohibitively restrictive when it comes to the recognition of cryptocurrencies as collateral (Basel Committee on Banking Supervision, 2022), creators of decentralized stablecoins can be much more creative as their coins usually fall outside the scope of banking regulation, at least for now. This allows them to explore new, innovative ways to stabilize the system. Generally, a fully transparent and well-designed blockchain-based stablecoin is inherently more stable than an opaque issuer-backed stablecoin that relies on traditional banking infrastructure.

An example of a not so well-designed stablecoin is the TerraUSD (UST), belonging to the class of *algorithmic stablecoins*. Before its spectacular collapse in May 2022, it was the third largest stablecoin in circulation (Liu, Makarov, and Schoar, 2023) (Briola et al., 2023). The TerraUSD was built on the Terra blockchain with a built-in oracle at the protocol level. The blockchain validators had to provide quotes for the supported currencies and were punished if they failed to do so or if their quotes were too far from the quotes of the other validators. The system's own token was the Luna, and the stabilization mechanism was based on a mechanism to print Luna tokens to buy TerraUSD in case it fell below the price of the reference currency, and to do the opposite if the price of the TerraUSD was too high. The stability of the TerraUSD rested on the assumption that the fundamental value of the Luna token does not fall too fast when more have to be printed to stabilize the system. Similarly to the currency of a country, this works well for as long as the economy around the Luna token, namely everything that happens on the Luna blockchain, is strong enough to support its value. However, it collapsed quite fast once the market lost trust in its stability. Another similarly designed token, the Iron Coin, faced the same fate. The Iron coin, which was supposed to track the USD dollar, was intended to be stabilized by printing Titanium tokens as needed. Generally, algorithmic stablecoins are considered risky as their value does not rest

[1]tether.to and circle.com/en/usdc
[2]bitcoinsuisse.com/cryptofranc

on tangible collateral (Clements, 2021). Nonetheless, some authors argue that algorithmic stablecoin can be *rational Ponzi schemes* if they can provide sound ways to credibly postpone their collapse indefinitely (Fu et al., 2023).

Besides issuer-backed and algorithmic stablecoins, there is a third class of stablecoins that is based on a blockchain-based collateral. Two of the most popular instances are the DAI and the Liquity USD (LUSD) (Chen, Fogel, and John, 2022; Lauko and Pardoe, 2021). Typically they let anyone deposit a collateral and then mint a certain quantity of the stablecoin. If the value of the collateral falls below a critical threshold, the deposit is liquidated and the proceeds are used to repay the minted coins. This is the principle that we apply to the Frankencoin, with the major difference being that the Frankencoin is much more adaptable to different collateral types, as it introduces a liquidation mechanism that does not rely on the presence of an oracle.

Problems with Oracles

An oracle is a service that records external observations on a blockchain, creating a data feed that is accessible to the smart contracts that reside on this blockchain. Typically, this data consists of prices observed in other places. When a decentralized stablecoins relies on a price oracle, this introduces an external dependency and potentially leads to centralization. For example, when analyzing the currently most popular oracle in the Ethereum system, Chainlink, one finds that its administrators could collude to manipulate prices and potentially exploit this capability to steal billions from decentralized protocols that rely on Chainlink. While the price feeds offered by Chainlink are the median of often more than a dozen independent sources, configuring the feed only requires the signatures of four of its administrators, enabling them to arbitrarily manipulate prices if they wish to do so.[3] But even if the administration was more decentralized, one would still have to trust the independent price sources to not collude. History shows that sometimes even the most reputable institutions cannot resist doing some price manipulation when the incentive to do so is big enough (Wheatley, 2012).

[3]For example, the Chainlink price feed for the ETH / USD price pair found at data.chain.link/ethereum/mainnet/crypto-usd/eth-usd is based on smart contract 0x5f4ec3df9cbd43714fe2740f5e3616155c5b8419. When we first looked into this smart contract in late 2021, only 3 out of 19 administrator signatures were required to configure the feed. After our observation made its way to the prolific crypto influencer Chris Blec (twitter.com/ChrisBlec), who prominently voiced his concerns on Twitter and Coindesk TV, Chainlink adjusted it to 4 out of 9 signatures. This is slightly better, but still a risk.

A more direct type of price feed is based on trades from decentralized exchanges that reside on the same blockchain. Here, an attacker would need to manipulate the prices through actual trades, which can be costly for popular assets traded on liquid markets, but less for illiquid assets. A further challenge is that a price feed chosen today might not be reliable any more in the future. Building a robust system based on the price-feed of a decentralized exchange would require a fallback mechanism in case liquidity on that exchange falls below an acceptable level, with that fallback in practice often boiling down to having a group of administrators that can configure the price source. That is why we came up with a new approach of price discovery that is tailored towards the situation at hand.

Governance

The launch of the Ethereum system made it possible to run programs on a blockchain. These programs are usually referred to as *smart contracts* and one of its first use cases was the creation of *decentralized autonomous organizations* (DAOs), that are governed by unstoppable code instead of human-driven processes (Jentzsch, 2016; Beck, Müller-Bloch, and King, 2018). However, the hacking of the first and at the time largest DAO (simply named *The DAO*) led to a shift away from complex governance mechanisms towards protocols that are as simple as possible and ideally even governance-free (Morrison, Mazey, and Wingreen, 2020; Ellinger et al., 2020).

In practice, this ideal is hard to attain and it is often desirable to have some minimally invasive form of governance to enable the protocol to evolve and adapt in a guided manner. Typically, this is done by the issuance of a transferable governance token that conveys voting rights that can be exercised in a democratic process. This section summarizes the governance process of two successful protocols, Uniswap and Maker, setting a benchmark for Frankencoin's governance.

Uniswap has a governance token called UNI that allows one to take part in a governance process based on majority votes. There are 1 billion UNI tokens in circulation, out of which about 200 million have registered themselves for voting by specifying a delegate. Delegates who command at least 2.5 million votes can create proposals. At the time of the proposal, a snapshot of the voting registry is taken and a voting period of 7 days starts. The proposal passes if it has at least 40 million supporting votes and if enjoys majority approval (Adams et al., 2021).

Out of the 41 proposals made so far, 28 have passed, 2 have failed, and 11 have been withdrawn again by the proposer. The ones that failed did so because they did not reach the 40 million quorum. No one seems to have ever attempted to make a malicious proposal. A potential attacker would have to invest more than 100 million to acquire 40 million UNI tokens, hope that the other tokens holders do not notice the proposal or do not take it seriously enough to bother with voting, and then make it pass in the last minute.

Maker employs a custom voting process they call 'top hat voting'. In this process, new proposals are automatically approved as soon as they received more votes than the previously most popular proposal. Furthermore, the votes of a participant move as they vote for a different proposal than they previously voted for, making it possible for the approval threshold to decline over time (Christensen et al., 2021).

Typically, it takes about 80000 to 90000 votes for a proposal to pass, which is about 9% of the MKR in circulation. That would also mark the amount of tokens needed to launch a successful attack. Potentially, a successful attacker might be able to get control over the system with an investment of roughly 100 million dollars at current market prices, allowing the attacker to mint arbitrary amounts of the DAI stablecoin if the attack is not detected early enough averted by a concerted effort of the major token holders.

Both the Uniswap and the Maker governance seem fit for their purpose. However, both require active participation for passing proposals and de facto control is exerted by only a handful of large token holders (Fritsch, Müller, and Wattenhofer, 2022). Also, the capacity of the system seems limited, as every proposal needs the attention of the major token holders and the potential for parallelization is limited.

2.1.2 Structure of the Chapter

This chapter is structured as follows. In section 2.2, the reader is introduced to the first cornerstone of the contribution, namely a method for oracle-free collateralized minting based on auctions. In Section 2.3, we introduce a reserve pool to address the residual risk of the system not being able to liquidate collateral at a price high enough to repay an open position. This pool is capitalized by the minters as well as voluntary contributors seeking a return on their Frankencoin holdings. At the same time, the reserve pool shares serve as governance tokens for the veto-based governance process described in section 2.4. Next, we turn our attention to how the peg to the Swiss franc is maintained in section 2.5. This completes the overall design of the Frankencoin, allowing us to outline the audited implementation in section 2.6

and concluding with an extensive risk analysis in section 2.7, in which we apply the typical capital requirements found in banking regulation to the system.

2.2 Oracle-free Collateralized Minting

This section specifies and analyzes the proposed oracle-free collateralized minting process from a game-theoretic perspective. There are two relevant games to be analyzed. The first is the initialization game between the minter and the system when proposing to mint new Frankencoins against a collateral. The second is the liquidation game to ensure the liquidation of positions that cease to be well-collateralized. The latter game has four participants: the minter, a challenger, bidders, and the system, whereas the challenger and the bidders are the actors that take the relevant decisions.

For simplicity, market prices are assumed to be constant for the duration of the games. Without loss of generality, the reference currency of the system is assumed to be the Swiss franc and the systems stablecoin is referred to as the Frankencoin or ZCHF. The calibration of the loan-to-value ratio l is subject to a separate analysis. It is not necessary to have a liquid market for the collateral asset. It suffices that some independent potential challengers owning sufficient quantities exist, such that the minter cannot corner the market. Finally, it is assumed that the market is efficient and that all actors are rational.

2.2.1 Definitions

Table 2.1 defines the four actors that play a role in the two games, the minter, the challenger, the bidder and the system. These actors have to choose strategies given the decision variables listed in table 2.2 and the exogenous parameters defined in table 2.3.

Letter	Name	Description
M	minter	Deposits a collateral of C_M and proposes a trigger price p_T.
C	challenger	Challenges the minter offering C_C units of the collateral.
B	bidder	Bids Z_B Frankencoins for C_C units of the collateral asset.
S	system	Vetoes new positions. Absorbs liquidation profits and losses.

Table 2.2: Actors. The four relevant actors in the two discussed games.

Letter	Name	Description
$C_M \in \mathbb{R}_{>0}$	minter collateral	Amount of the collateral provided by the minter.
$p_T \in \mathbb{R}_{>0}$	trigger price	Price level below which the position is liquidated.
$C_C \in (0, C_M]$	challenger collateral	Collateral asset quantity offered by the challenger.
$p_B \in \mathbb{R}_{>0}$	bid	The price the bidder is offering.
$D_S \in noop, veto$	governance	The system's approval decision.

Table 2.4: Strategies. The decision variables in the two discussed games.

Letter	Name	Description
$l \in [0, 1]$	loan-to-value	The usable fraction of the collateral value.
$p \geq 0$	price	The market price of the collateral in Frankencoin.
$k \in [0, 1 - l)$	reward	Fraction of the bid to reward successful challenges.
$v > 0$	value	The utility value the minter derives from a successful minting.

Table 2.6: Parameters. The relevant parameters and their meaning.

2.2.2 Initialization Game

The initialization game is about initiating a new collateralized Frankencoin position.

Specification

Two actors, a minter M and the system S take part in this sequential game with two rounds as depicted in Figure 2.1. In the first round, the minter chooses a

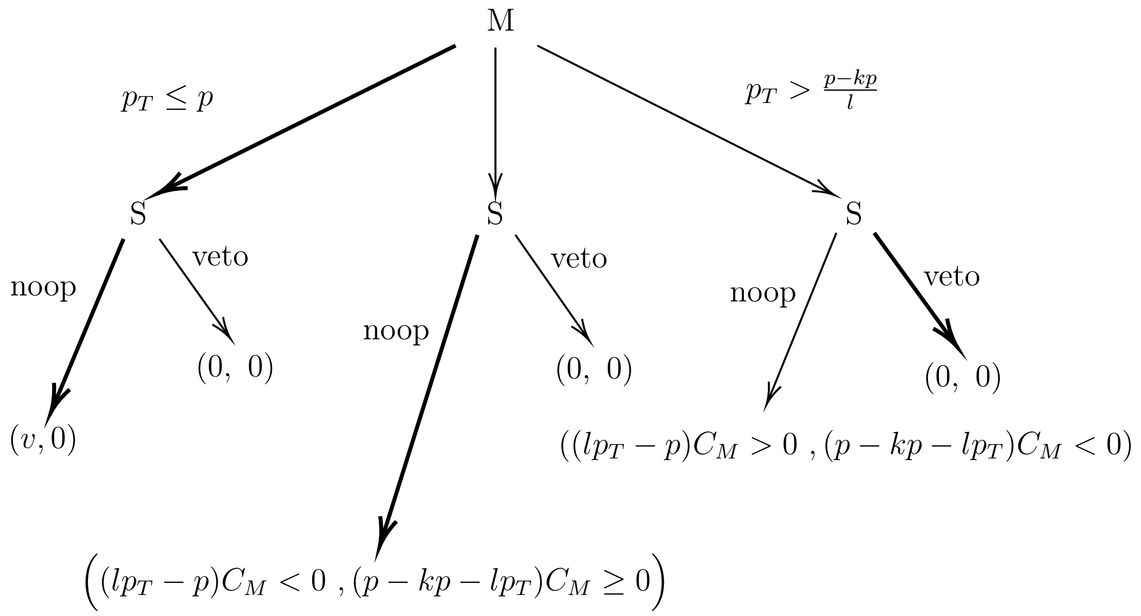

M
$p_T \leq p$
$p_T > \frac{p-kp}{l}$
S
S
S
noop
veto
noop
veto
noop
veto
$(0,\ 0)$
$(0,\ 0)$
$(0,\ 0)$
$(v, 0)$
$((lp_T - p)C_M > 0\ , (p - kp - lp_T)C_M < 0)$
$\Big((lp_T - p)C_M < 0\ , (p - kp - lp_T)C_M \geq 0\Big)$

Figure 2.1: Initialization game. Extensive form graph of the initialization game between the minter (M) and the system (S), with the preferred choices in bold. The first element in the round brackets is the minter's payoff and the second element that of the system. A rational minter M goes for the leftmost path, choosing a liquidation trigger p_T below the market price, allowing them to successfully mint some Frankencoins and derive utility v from the obtained liquidity. In the other two cases, the minter is either liquidated in the subsequent liquidation game and suffers a loss, or the position is so undercollateralized that it is immediately vetoed by the system to avert a loss.

strategy $(C_M, p_T) \in \mathbb{R}^2_{>0}$. The variable C_M is the amount of the collateral that the minter deposits, and the variable p_T is the price point at which the liquidation is triggered. The trigger price also determines how many Frankencoins the minter can mint and withdraw, namely up to $Z_M = lp_TC_M$. In the second round, the system chooses a strategy $D_S \in \{noop, veto\}$, whereas *noop* is a common abbreviation for 'no operation' and represents the choice of doing nothing. The other possibility is to *veto* the proposal.

It is assumed that the minter has a desire to borrow some Frankencoins while retaining ownership of the provided collateral. This is represented by a utility of value v in case of a successful initialization of a well-collateralized position. Since the minted Frankencoins and the resulting debt are of equal magnitude, there is no benefit for the minter besides v. A position is considered *well-collateralized* as long as $p_T \leq p$. On the two depicted paths where the position is not well-collateralized, the payoffs are determined by the subsequent liquidation game. The two relevant ranges to consider are the one where the collateral suffices to cover the loss, such that the system benefits from the liquidation, i.e. $p_T \leq \frac{p-kp}{l}$, and the one where the position is so under-collateralized that the liquidation leads to a loss for the system.

Analysis

Theorem 1 (Valid Initialization). *Assuming the liquidation game is successful at liquidating undercollateralized positions and yields the payoffs from table 2.5 (Theorem 2), the initialization game will never end with a position being opened that is not well-collateralized, whereas well-collateralized is defined as $p_T \leq p$.*

Proof. Theorem 1 can be shown by analyzing the initialization game as depicted in Figure 2.1.

Case 1 (left path, $p_T \leq p$). *The collateralization is sufficient to prevent a liquidation. The system S is indifferent between veto or noop, as there is no loss or gain for either decision. The resulting position, if any, is well-collateralized.*

Case 2 (middle path, $p < p_T \leq \frac{p-kp}{l}$). *In this case, the position is not well-collateralized. The system has to choose between vetoing it and doing nothing. Doing nothing leads to the successful initialization of the position and, assuming that Theorem 2 holds, a liquidation with a positive payoff for the system and a negative payoff for the minter. The minter would get away with up to $Z_M = lp_TC_M$ Frankencoins, but would also suffer a loss of $pC_M > Z_M$. This leads to a negative payoff and the conclusion that no rational minter will ever choose p_T in the range $p < p_T \leq \frac{p-kp}{l}$.*

Case 3 (right path, $p_T > \frac{p-kp}{l}$). *In this case, the position is so under-collateralized that a liquidation would yield a negative outcome for the system after auctioning off the collateral and paying the challenger reward. With a veto, the system S can avert that loss, resulting in no position being opened and a payoff of 0 for the minter.*

To conclude, no rational minter will ever propose an position that is not well-collateralized. □

Extensions

In practice, casting a veto comes at a cost for the system as someone needs to invest time into reviewing the proposed position, estimating the market price of the collateral, and potentially casting a veto. To counter this, the actual system imposes an application fee f on the minter when applying for a new position. Furthermore, it is possible for the system to impose an interest $i < v - f$ without deterring the minter from making a proposal. For simplicity, we do not formalize these additional factors.

2.2.3 Liquidation Game

The purpose of the liquidation game is to liquidate undercollateralized positions after the market price has fallen below the trigger price, i.e. $p < p_T$. The payoffs are designed such that the challengers have an incentive to start the challenge only when the position actually is undercollateralized.

The liquidation game consists of two stages: a decision to challenge the collateralization of a position and the subsequent competitive decisions to bid on the challenge. Anyone can start a challenge and thereby become the challenger. Also, anyone can place a bid. In the base scenario, it is assumed that the minter, challenger, bidder and system are independent. In subsequent sections, we show that the game still works as intended if identities overlap, for example when the bidder and the minter are the same person.

Specification

Two actors, the challenger and a group of competitive bidders take part in a sequential game as depicted in Figure 2.2. The minter is not allowed to repay the outstanding balance and reclaim the collateral for as long as a challenge is pending and is therefore not an actor in this game. Neither is the system.

The liquidation game starts with the challenger's decision to start a challenge with quantity $C_C \leq C_M$ of the collateral. Then, it is the bidder's turn to bid for that quantity of the collateral and to choose a bid $Z_B = p_B C_C$, implying the bidding price p_B. In practice, the bidding process is conducted as a Dutch auction. The price starts at p_T for a while and then starts to linearly decline towards 0 until a bidder finds the offer attractive enough. The price point at which this happens is denoted p_B. Given a competitive process and an efficient market, the sale will either happen at p_T or the market price p, i.e. $p_B = min(p, p_T)$.

Challenges that end with $p_B < p_T$ are considered successful. They imply that the position is not well-collateralized. Challenges that end with $p_B = p_T$ are considered averted. In the successful case, it is C_C of the minter's collateral that the bidder is bidding for. In the averted case, the bidder will get the challenger's collateral. The switch in the bidding target is the key element of the Frankencoin auction.

In the averted case, the bidder ends up buying C_C units of the collateral asset from the challenger for $p_B C_C = p_T C_C$, with the other actors being unaffected. In the successful case, the bidder ends up buying the same quantity of the collateral from the minter, with the proceeds going to the system, the minter's associated debt $l p_T C_C$ being erased, and the challenger being rewarded with $k p_B C_C$. These payoffs are shown in Table 2.4. and illustrated in Figure 2.3 for the case of the successful challenge.

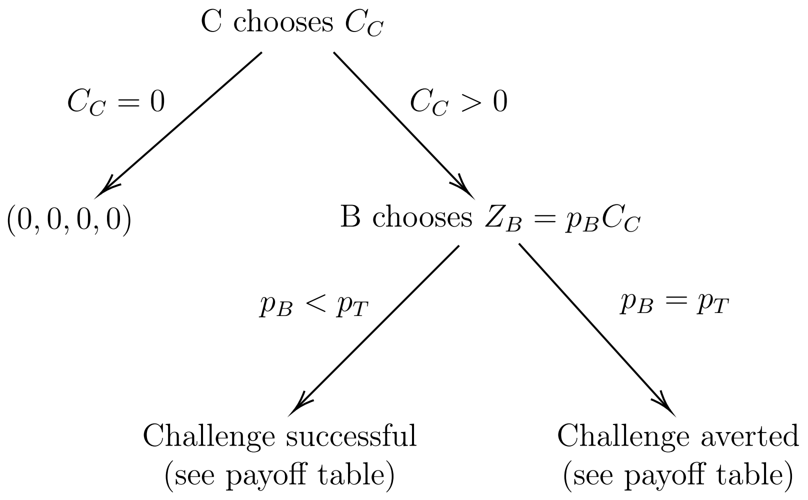

C chooses C_C
$C_C = 0$
$C_C > 0$
$(0, 0, 0, 0)$
B chooses $Z_B = p_B C_C$
$p_B < p_T$
$p_B = p_T$
Challenge successful (see payoff table)
Challenge averted (see payoff table)

Figure 2.2: Liquidation game. Extensive form graph of the liquidation game. It only makes sense to start a challenge if the highest bid can be expected to imply a violation of the loan-to-value threshold, i.e. if the market price of the collateral is below the trigger price.

Base Scenario

In the base scenario, it is assumed that the minter, challenger, bidder and system are distinct persons.

Theorem 2 (Successful Liquidation). *Given rational actors, a challenge is started if and only if the market price has fallen below the trigger price, i.e.* $p < p_T$*, and the challenge will end successfully.*

Actor		Challenge successful	Challenge averted
B	bidder	$p - p_B$	$p - p_B$
C	challenger	kp_B	$-p + p_B$
M	minter	$lp_T - p$	0
S	system	$p_B - kp_B - lp_T$	0

Table 2.8: Payoffs. The payoff for each actor in the liquidation game in relative terms. To get the absolute payoff, all values need to be multiplied by the size of the challenge C_C.

Proof. The payoff of the bidder does not depend on whether the challenge is successful or averted, making it perfectly predictable. Being rational, the bidders will never bid higher than the market value, i.e., $p_B \leq p$. Considering that the Dutch auction starts at price p_T, the highest possible bid is $p_B = p_T$. Under perfect competition in an efficient market, the highest bid therefore is:

$$Z_B = p_B C_C = \min(p, p_T) C_C \tag{2.1}$$

Predicting the behavior of the bidder, the challenger immediately start a challenge once the market price has fallen below the trigger price, i.e. $p < p_T$, allowing them to earn the challenger reward $kp_B C_C$. In contrast, they will not start a challenge when $p \geq p_T$ as the payoff would be $p_B - p = p_T - p \leq 0$.

Therefore, given the specified payoffs, rational challengers will start a challenge as soon as $p < p_T$ (but not earlier) and that challenge will be successful. $\square$

Overlapping Identities Scenario

A key assumption of the base scenario was that all actors are independent. However, in practice, the bidder and the minter might be the same person, or there might be side-payments between them to tilt the incentives. This section shows how theorem 2 is affected and how it can be preserved when the bidder does not act on its own. The other three conceivable combinations (i.e. challenger and system as the same person) are less problematic and not further discussed.

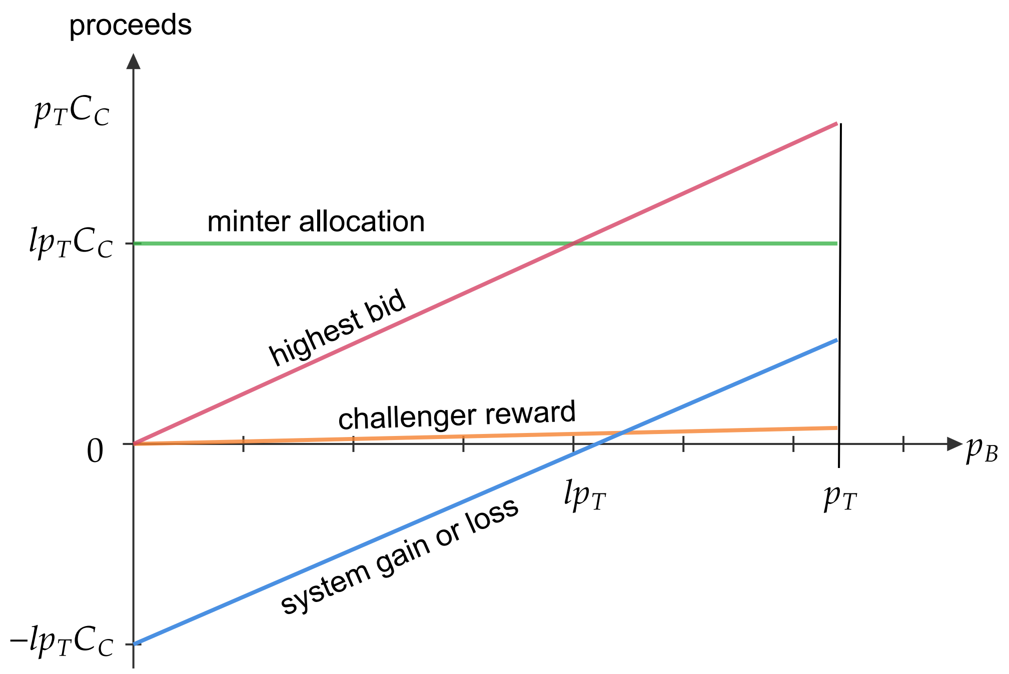


Figure 2.3: Bid allocation after a successful challenge. How the proceeds from the winning bid (red) are allocated to the minter (green), the challenger (orange), and the system (blue), as a function of the bidding price p_B. The four lines sum up to zero. It is pivotal for the system to ensure that challenges happen before the price has fallen too low in order to avoid losses. In the extreme case of the collateral having become worthless, the system has no choice but to forgive the outstanding amount lp_TC_C.

Proposition 1 (The challenger as bidder)**.** *Theorem 2 still holds when allowing the challenger to be the bidder.*

This proposition is shown by distinguishing the case of the successful from the case of the averted challenge.

In case of a successful challenge, $p < p_T$ holds by definition. If that was not the case, other bidders would bid $p_B = p_T$ and avert the challenge. When the highest bidder and the challenger are the same person, the cumulative payoff for the successful case is:

$$p - p_B + kp_B > 0$$

In case of an averted challenge, the cumulative payoff of the bidder and the challenger is:

$$p - p_B - p + p_B = 0$$

This implies that a challenger can cancel a challenge at any time without incurring any costs, simply by bidding $p_B = p_T$. It also implies that the challenger has no incentive to artificially avert an otherwise successful challenge.

Furthermore, within the successful challenge case, the challenger has no incentive to make a bid above the market price ($p_C > p_B = p$) as the payoff without interference is larger than the payoff with interference ($kp_B > p - p_C + kp_B$).

Therefore, a rational challenger will not bid any different than any other bidder in case of $p < p_T$, and would not start a challenge otherwise, preserving theorem 2.

Proposition 2 (The minter as bidder)**.** *Theorem 2 is violated when allowing the minter to be the bidder in an isolated game, but still holds when allowing the challenger to repeat the game under the same market price p.*

The minter might want to avert a looming liquidation by placing bids above the market price, i.e. $p_B = p_T > p$. Bidding above the market price to avert a challenge yields a negative payoff of $p - p_B < 0$. However, taking this loss might still be preferable over taking the liquidation loss of $lp_T - p$. So in such a case, a rational minter would bid above the market price, leading to the challenge being averted despite $p < p_T$, violating theorem 2!

To address this problem, one needs to expand the scope from a one-shot game to a repeated game. When the minter averts the challenge by bidding above the market price with $p_B = p_T > p$, the challenger makes a profit of $p_B - p > 0$. If the challenger is given the opportunity to restock the auctioned asset at market price p and then repeat the game before the minter gets a chance to close the open position, theorem 2 still holds. In that case, the challenger can threaten to infinitely repeat the game, causing an infinite loss to the minter. Under this threat, a rational minter reverts to the original bidding strategy under which the ideal bid is the same as for the other bidders.

Proposition 3 (The system as bidder)**.** *Theorem 2 is violated when allowing the system to be the bidder in an isolated game, but still holds when allowing the challenger to repeat the game under the same market price p.*

This is a somewhat theoretic scenario as the system is governed by commonly agreed rules and cannot act arbitrarily. Nonetheless, it is worth analyzing for completeness. To analyze this scenario, we distinguish three cases, one in which the challenge is already averted without the system's intervention, one in which the system overbids to avert a challenge, and one in which the system places a bid such that $p < p_B < p_T$.

In case of the market price being high enough that other bidders already can be predicted to avert the challenge, the incentive of the system acting as bidder are identical to that of the other bidders, as the payoff for the system in that case is zero.

For the other two cases, the cumulative payoff of the system for a successful challenge needs to be considered. It is:

$$p - p_B + p_B - kp_B - lp_T = p - kp_B - lp_T$$

This payoff is identical to the payoff without intervention with $p = p_B$. In the case of a positive payoff, the system has no incentive to deviate from the standard behavior. In the case of a negative payoff, the system could be tempted to bid artificially high like the minter in proposition 2 to avert the challenge and to postpone the looming loss. However, this is not sustainable in a repeated game and even in a one-shot game one could argue that the system did not improve its situation as it still sits on the same undercollateralized loan.

Therefore, the outcome remains the same and the validity of Theorem 2 is preserved.

2.2.4 Example

As an example, let us say the minter opened a position with 10 ABC tokens at a trigger price of $p_T = 100$ as collateral and that $l = 0.8$ and $k = 0.02$. The minter withdraws 800 ZCHF. Then, a challenge is started with 4 ABC tokens, and the highest bid ends up being 360 ZCHF, so the challenge is successful since $360 < 400$. The bidder gets 4 ABC tokens from the minter. The minter's debt is reduced by 320 ZCHF to 480 ZCHF backed by 6 ABC tokens. From the 360 ZCHF bid, 320 ZCHF are burned, $k * 360 = 7.2$ ZCHF are given to the challenger as a reward, and the remaining 32.8 ZCHF are assigned to the reserve as profits for the system.

2.2.5 Minter Reserve

The Frankencoin system has three lines of defense that guard against an undercollateralization of the system. The first line of defense is the overcollateralization of the individual positions as defined by the parameter l. The second line of defense is the equity capital of the system as described in the subsequent chapter. And the third line of defense is the use of other positions in case one position fails through a mechanism denoted as the *minter reserve*. This allows the system to burden the minters with additional debt to bring the books back into balance and should be seen as a measure of last resort once everything else has failed.

To feed the minter reserve, the minting mechanism is adjusted and two additional global variables introduced, the actual reserve R and the target reserve $\tilde{R}$. Given a position with C_M and p_T, the minter will still be allowed to mint and withdraw up to $Z_M = lp_TC_M$ Frankencoins, but at the same time an additional $Z_{R,t} = (1-l)p_TC_M$ Frankencoins are minted and added to the reserve on behalf of the minter. When the minter repays the position, the reserve contribution is fully returned again under normal circumstances, but might be lower in stress scenarios.

When adding $Z_{R,t}$ to the reserve at time t, the actual reserve and the target reserve are updated as follows:

$$R_t = R_{t-1} + Z_{R,t}$$
$$\tilde{R}_t = \tilde{R}_{t-1} + Z_{R,t}$$

Including the reserve, the de facto total debt of the minter at time t is $Z_T = Z_M + Z_{R,t}$. In order to close position at time s, the minter must repay:

$$Z'_M = Z_T - \frac{R_s}{\tilde{R}_s} Z_{R,t}$$

As long as $R_s = \tilde{R}_s$, this is identical to the scenario without the minter reserve. However, they might have to repay more in case the system suffered from a loss that could not be covered by equity capital. In case of a loss L_t at time t that cannot be covered by the first two lines of defense, the missing Frankencoins are taken out of the minter reserve, implicitly increasing the amount of Frankencoins every minter needs to return in order to get their collateral back:

$$R_t = R_{t-1} - L_t$$

After a loss has happened, the reserve will be below its target, i.e. $R < \tilde{R}$ and minters will have to return $Z'_M > Z_M$ to close their positions. As long as the system reserve is below its target, all capital flows that normally go into the equity capital of the system are redirected to the minter reserve to replenish it.

Effectively, this also leads to a temporarily higher loan-to-value ratio $\tilde{l}$ when reasoning about the stability of a position:

$$\tilde{l} = (l - 1)\frac{R}{\tilde{R}} + 1 \geq l$$

This new loan-to-value ratio $\tilde{l}$ only applies when repaying an outstanding amount. When minting new Frankencoins, the old l is still applicable.

So in a distress scenario, a minter might for example mint 80 ZCHF but would need to repay 81 ZCHF to close the position again with $l = 0.8$ and $\frac{R}{\tilde{R}} = 0.95$. So as long as the minter reserve is not replenished, closing a position will come with an additional implied fee.

This change has no qualitative impact on the initialization and the liquidation game, although it temporarily makes the initialization more costly until the minter reserve is replenished.

2.3 Equity and Pool Shares

The Frankencoin offers different functions that are fulfilled by commercial banks in the off-chain economy. It offers a fungible means of payment and it allows its users to borrow money against a collateral. Furthermore, it can suffer from losses and generate profits. These losses and profits accumulate in a pool that belongs to the holders of Frankencoin Pool Shares (FPS).

Frankencoin Pool Shares are freely transferable tokens that resemble the shares in a company. Formally, they do not fulfill the requirements to be a security. But functionally, they let the holders participate in the system's economic success (or failure). Like traditional shares, they are also the basis for participation in the governance process.

2.3.1 Issuance and Redemption

Issuance and redemption of pool shares are governed by the principles of the *Continuous Capital Corporation* presented in chapter 3. At any time, new investors can contribute additional capital and get Frankencoin Pool Shares in return. And existing shareholders can redeem their shares at the same price as determined by the pricing rule of the continuous capital corporation. To derive this pricing rule, one needs to specify its production function first.

There is extensive literature on the production function of the banking sector (Benston, Hanweck, and Humphrey, 1982; Gilbert, 1984; Clark, 1984). However, it is unclear how to apply the existing insights to the Frankencoin system. While the early research often assumes Cobb-Douglas production, which would make it easy to derive the pricing function of the continuous capital corporation, their insights are often not directly applicable. For example, Bell and Murphy (1968) use a Cobb-Douglas production function, but their measure for the bank's output is the number of accounts, whereas we are looking at how monetary income depends on the employed capital. Therefore, we resort to just assume Cobb-Douglas production for simplicity and then show that the chosen parameters lead to a sensible result given the circumstances.

For the purpose of deriving the pricing rule, we focus on capital as the only input, arriving at a production function of the form:

$$f(K) = AK^{\alpha}$$

The exponent α represents the *capital share* (as opposed to the labor share that is skipped here) in macroeconomic models. It denotes how much of the system's value stems from its capital and typically floats between 20% and 40% for the economy as a whole (Ilo, 2015). The factor A represents the level of productivity or technology, but cancels itself out in the subsequent calculations.

With the economic consequences presented in section 2.3.3 in mind, we have chosen a value of $\alpha = 1/3$ for the Frankencoin system, leading us to a market cap or valuation of:

$$V(K) = \frac{f(K)}{f'(K)} = 3K$$

Given the number of shares in circulation s, the price of one FPS is given by $p(K,s) = \frac{3K}{s}$.

To calculate the number of shares an investor gets by contributing ΔK to the reserve, we calculate the number of shares s_{t+1} after the investment given the number of shares s_t before the investment using the capital-dependent function 3.9 from Chapter 3 for the number of shares $\theta(K)$ as follows:

$$s_{t+1} = \frac{\theta(K+\Delta K)}{\theta(K)} s_t = \frac{f(K+\Delta K)}{f(K_0)} \frac{f(K_0)}{f(K)} s_t = \left(\frac{K+\Delta K}{K}\right)^{\frac{1}{3}} s_t$$

The advantage of looking at the relative increase of the number of shares s is that this equation is still valid in the presence of other factors that influence K between transactions, namely incoming profits and outgoing losses. In contrast, the static equation for $\theta(K)$ derived in chapter 3 is only generally valid when profits and losses are immediately attributed to the shareholders, like it is done in general equilibrium models, but not in reality.

2.3.2 Frontrunning and Remedy

One of the risks of the continuous capital corporation is that it opens the door for frontrunning at the cost of the shareholders. This risk is particularly accentuated for systems running on a public ledger where imminent flows of capital can be anticipated and acted on before they are processed by the system. For example, if a profitable liquidation of a large position is imminent, it would be profitable to buy some pool shares immediately before the liquidation and redeem them again immediately afterwards in a so-called sandwich attack, allowing the attacker to make a nice return in a short time at the expense of the other shareholders. The Frankencoin system guards against this risk by enforcing a minimal holding period and a transaction fee.

To quantify the profits that can be extracted through such a sandwich trade, consider how the number of shares changes during the attack, with the attacker first investing i, then the system receiving a profit π, and finally the attacker divesting d again:

$$s_{t+2} = \left(\frac{K+\pi+i-d}{K+\pi+i}\right)^{\frac{1}{3}} s_{t+1} = \left(\frac{K+\pi+i-d}{K+\pi+i}\right)^{\frac{1}{3}} \left(\frac{K+i}{K}\right)^{\frac{1}{3}} s_t$$

Here, s_t denotes the number of shares in circulation before the trade, s_{t+1} after investment i, and s_{t+2} after the trade. Assuming that the attacker wishes to do a neutral attack and end up with the same number of shares as they started, $s_{t+2} = s_t$ holds. With this assumption, the above equation can be solved for d in order to quantify the proceeds from the divestment $d(i)$ as a function of the investment i.

The attacker wishes to maximize their profits by choosing i:

$$\max_i d(i) - i = K + \pi - \frac{(K + i + \pi)K}{K + i}$$

The larger the employed capital i, the larger the amount the attacker can extract, with:

$$\lim_{i \to \infty} d(i) - i = \pi$$

With infinite Frankencoins, a frontrunner could potentially reap all the imminent system profits for themselves in a sandwich attack. Such a sandwich attack would consist of a transaction consisting of a large flash loan, buying as many shares as possible, triggering the profitable event (i.e. the end of an auction), selling the shares again, and finally repaying the flash loan. In a scenario in which the Frankencoin system itself supports native flash loans, such loans could potentially be much larger than the amount of ZCHF in circulation.

The Frankencoin system averts such attacks and discourages short-term speculation in general by enforcing a minimal holding period of 90 days. However, this measure on its own still leads to undesired arbitrage opportunities where a minority of active traders can extract a profit at the expense of the passive investors. The associated price dynamics are illustrated in figure 2.4.

Having small-scale arbitrageurs extract small profits from the system whenever there is a small inflow or outflow of equity capital does not seem to add value to the system. Therefore, we seek to prevent it by introducing a price spread of 0.6%, which is equivalent to a 0.3% fee and in line with the standard fees of the Uniswap system. A price spread is a common method to protect against losses when trading with informed traders (Glosten and Milgrom, 1985).

Another way to guard against such attacks would be to smoothen incoming profits and losses. This would make the attack more costly as the attacker would not only need to deploy capital for an atomic instant, but would need to employ it over a longer period of time. However, such a smoothing would cause the pricing function

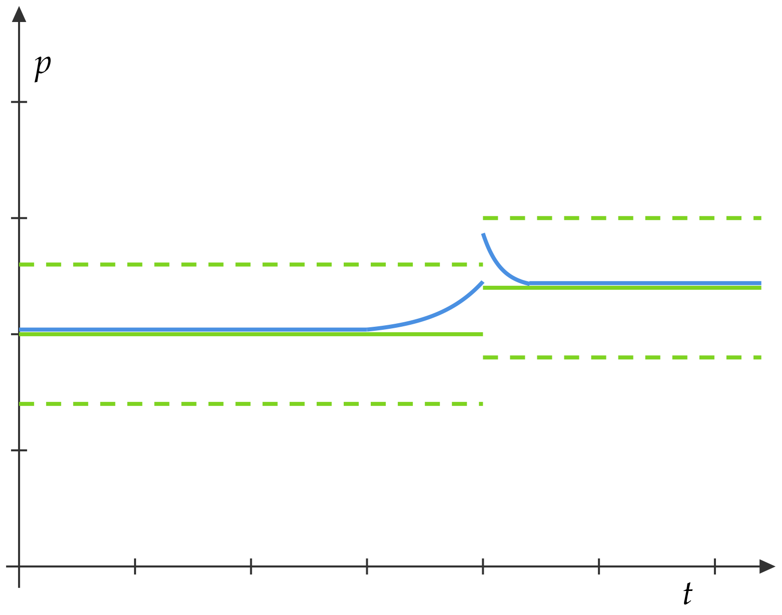


Figure 2.4: Stylized anticipated price change. In the absence of any trading, the inflow of a profit leads to an immediate, proportionate adjustment of the price at which pool shares can be obtained or redeemed (green line). In an active market, traders anticipate this change and start buying before the change, and sell again after the change (blue line), making a profit at the expense of the passive shareholders. Introducing a price spread (dashed green line) prevents intertemporal arbitrage for small price changes that happen at discrete steps.

to exhibit a lag. While the trading fee also leads to a lagged price discovery, the mispricing caused by the fee is limited relative to the price (the 0.3%), whereas the mispricing introduced by profit smoothing is only bound in time and can potentially be much larger as changes accumulate.

2.3.3 Equilibrium Reserves

Assuming similar risk premia for debt in the Frankencoin system and pool shares, the economic equilibrium is reached when about 1/3 of the effectively borrowed Frankencoins are in the reserve pool.

To arrive at this result, we disregard the minters reserve and assume that the interest paid on the effectively usable part D of all open Frankencoin debt positions is r_d, whereas $D = O - R$ is defined as the total amount of Frankencoins minted against a collateral O minus the minters reserve R. The choice of letter O is a reference to the equivalent label minter repayment obligation from appendix 2.A.

Further, let us denote the expected return of holding pool shares as r_e. Assuming an equivalent risk premium, this leads to an arbitrage opportunity whenever $r_d \neq r_e$. If $r_d < r_e$, minters can borrow more and invest it into pool shares, thereby driving up the price. If $r_d > r_e$, investors have an incentive to sell pool shares instead of increasing or renewing their debt positions.

We define r_d to contain not only the nominal interests, but also fees, liquidation proceeds, liquidation losses, and all other income streams directly or indirectly attributable to the minters. Then, one can expect in equilibrium $r_d D = r_e V(K)$ given the aforementioned arbitrage opportunities. The interest paid on the sum of debt positions D corresponds to the return one can expect on investments in pool shares with the market capitalization of the pool shares being $V(K) = 3K$ as defined in section 2.3.1. This implies $K = \frac{1}{3}D$ in the static equilibrium. The effective ratio can vary in times of changing interest rates as there is no incentive to cancel already established positions early and no mechanism to adjust the interest rates of established positions.

This, in turn, also implies that in equilibrium, $\frac{2}{3}D$ ZCHF are used for other purposes, for example as a transactional currency, and that the Frankencoin system does not pay any interest to their holders. The underlying assumption is that there is sufficient demand for a stablecoin as a transactional currency such that the system as a whole can reap some seignorage gains and the early investors in Frankencoin Pool Shares get a return on their capital that significantly exceeds r_e. Considerations for what happens if there is a mismatch between the demand for Frankencoins as a transactional currency and the demand for loans are made in section 2.5.

2.4 Governance

The Frankencoin system is designed for passive, decentralized governance. Under the assumption of perfect information and rational participants, no governance action will ever be necessary. This is achieved by having a passive system based on vetoes that only require an intervention if things go wrong. Having a credible threat of intervention can elicit correct behavior even if the intervention actually never takes place.

Most decentralized autonomous organizations build on a vote-based governance system with some form of majority votes as discussed in 2.1.1. In contrast, the

Frankencoin system is based on vetoes. Its speed and the reduced amount of required attention make veto-based governance more agile, faster, and more light-weight in comparison to a full-scale majority vote. However, a veto-based system comes with the disadvantage of a small minority being able to block the whole system. This is averted by time-weighting the votes and the introduction of a 'kamikaze' function that allows an altruistic token holder to reduce the voting power of an attacker.

In the following, we specify the governance process, derive how a smart contract can keep track of the voting weights of each shareholder, and show that the governance of the Frankencoin system yields the same outcome as a majority vote while at the same time requiring fewer interactions than traditional voting schemes.

2.4.1 Specification

The governance system is subject to the following rules:

1. Anyone can make proposals. Making a proposal costs a fee of c_p.
2. Proposals pass after t_p days unless someone vetoes them.
3. Anyone with more than $q = 2\%$ of the total votes V has veto power, i.e. a user with v votes can veto if $v > qV$ holds.
4. The number of votes of a user is calculated by multiplying their Frankencoin Pool Shares with the time they have held them.
5. Users can delegate their votes to other users, who in turn can delegate them further. This allows minority shareholders to team up for a veto.
6. Users can persistently cancel each others votes. For example, Alice can sacrifice 100 votes in order to also reduce Bob's number of votes by 100.

Having time-weighted votes shifts power towards users with a long-term interest in the system. In particular, it guards against malicious actors that take flash loans or use other short-term instruments to temporarily get a sufficiently large number of shares in order to sabotage the system by vetoing good proposals.

Figure 2.6 shows the decision tree with the decision points for the supporters and the opposition of a proposal. It is more deeply analyzed in section 2.4.3. There, we prove that the outcome of this veto-based process is equivalent to that of a majority-based voting process, show how both approaches differ when taking costs into account, and finally argue that the veto based approach is more efficient.

2.4.2 Vote Accumulation

To be able to determine whether a given token holder has more than 2% of all votes, the system does not only need to keep track of each individual user's vote, but also of the total number of votes. Since computations and storage slots on the blockchain are expensive, it is important to keep track of the votes as well as the total in an efficient way. This section specifies how this is done in the Frankencoin system, namely by storing anchor time stamps that are adjusted when tokens are moved. This method requires one additional stored variable per user as well as two global variables to keep track of the total. All operations are performed in constant time, i.e. in O(1) according to the nomenclature of computer science.

The calculation of the votes of a given user is based on that user's balance and a time anchor. Denoting a_s the anchor of a user at discrete time step s and b_s the token balance at the same time, the number of votes at continuous time t can be defined as:

$$v(t) = b_{s(t)}(t - a_{s(t)})$$

Here, $s(t)$ is a function that maps continuous time onto the number of balance changes that have happened at this point in time. For every point in time t at which a balance change happens, s increases by one. Further, let us denote t_s as the time at which balance change s happened and t_{s-} the point in time immediately before that, i.e. $t_s = t_{s-} + \epsilon$ for an arbitrary small $\epsilon > 0$. Consequently, $s(t_s) = s = s(t_{s-}) + 1$.

$$a_s = \begin{cases} a_{s-1}, & \text{for } b_s \leq b_{s-1} \\ t_s - \frac{v(t_{s-})}{b_s} & \text{otherwise} \end{cases}$$

Proposition 4 (Anchor updates)**.** *The anchor update rule exhibits the required properties, namely that votes are adjusted downwards proportionally when the user's balance declines and stay constant when the balance increases.*

Proof. When the balance of a user declines in step s, the number of votes declines proportionally:

$$\frac{v(t_s)}{v(t_{s-})} = \frac{b_s(t_s - a_s)}{b_{s-1}(t_{s-} - a_{s-1})} = \frac{b_s}{b_{s-1}}$$

When a user receives tokens, the votes stay the same:

$$v(t_s) = b_s(t_s - a_s) = b_s(t_s - t_s + \frac{v(t_{s-})}{b_s}) = v(t_{s-})$$

.

□

To keep track of the total votes $V(t)$ in the system, two further variables are needed. These are the total vote anchor A_s and the total vote anchor timestamp T_s. Whenever a number of votes l_s have been discarded in step s, these variables need to be updated as follows:

$$A_s = V(t_{s-}) - l_s$$

$$T_s = t(s)$$

This ensures that the total number of votes can always be calculated as

$$V(t) = A_{s(t)} + B_{s(t)}(t - T_{s(t)})$$

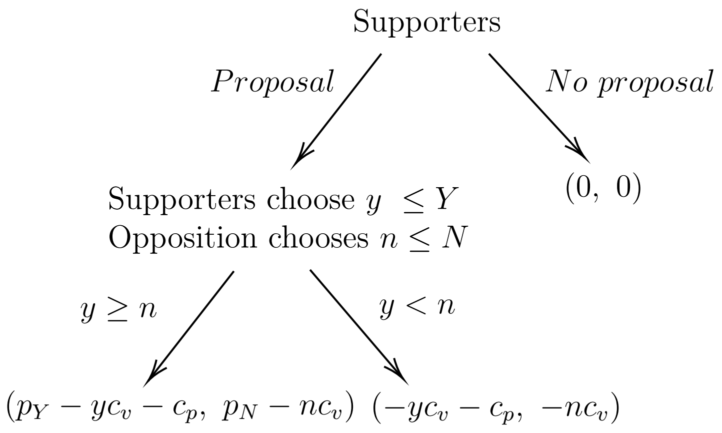
with B_s denoting the total token supply at step s.

Continuously updating the vote counts on each transaction comes at a cost and makes the transfer of pool share tokens about twice as expensive as a plain token. Other governance tokens, for example those of Uniswap or the Maker protocol, address this by only tracking the votes of those tokens that registered themselves for voting. At the same time, they come with more heavy-weight voting mechanisms that rely on snapshots of the token register.

2.4.3 Outcome Equivalence

This section shows that the Frankencoin governance system yields the same outcome as a majority vote with rational participants. Let us denote Y the number of votes in favor of a proposal, and N the votes against (regardless of whether they have been cast or not in an actual voting process).

Our benchmark is a standard majority vote that accepts the proposal when $Y > N$ as depicted in figure 2.5 with payoff $p_Y > 0$ for the supporters and payoff $p_N < 0$ for the opposition, and voting costs $c_v > 0$ per vote.



Supporters
Proposal
No proposal
(0, 0)
Supporters choose $y \leq Y$
Opposition chooses $n \leq N$
$y \geq n$
$y < n$
$(p_Y - yc_v - c_p,\ p_N - nc_v)$
$(-yc_v - c_p,\ -nc_v)$

Figure 2.5: Vote game. With profits $p_Y > 0$ for those in favor, loss $p_N < 0$ for those against, voting costs $c_v \geq 0$ per vote, and proposal costs $c_p \geq 0$. Once the supporters decided to make a proposal, the Y supporters and the opposition of size N simultaneously choose how many votes y and n to cast.

An extensive form graph of the Frankencoin's governance process is shown in figure 2.6. In addition to the variables from the benchmark game, there is a value k to denote the number of mutually cancelled votes in case the supporters choose to do so in order to prevent the opposition from vetoing the proposal in future attempts. Also, in this case c_v stands for the costs of casting a veto and not the costs for casting a vote.

Theorem 3 (Outcome Equivalence)**.** *Given rational actors and disregarding participation costs, the supporters of a proposal can get it approved if and only if it would also pass in a simple majority vote, i.e. in both the vote game and the veto game, proposals will get through if and only if $Y > N$ when $c_v = c_p = 0$ with veto quorum $q \in (0, 1]$.*

Proof. For the base case of the majority vote without costs, supporters will choose $y = Y$, and the opposition will choose $n = N$ and the proposal will pass whenever $Y > N$. What is left to prove is that the same applies to Frankencoin's veto based

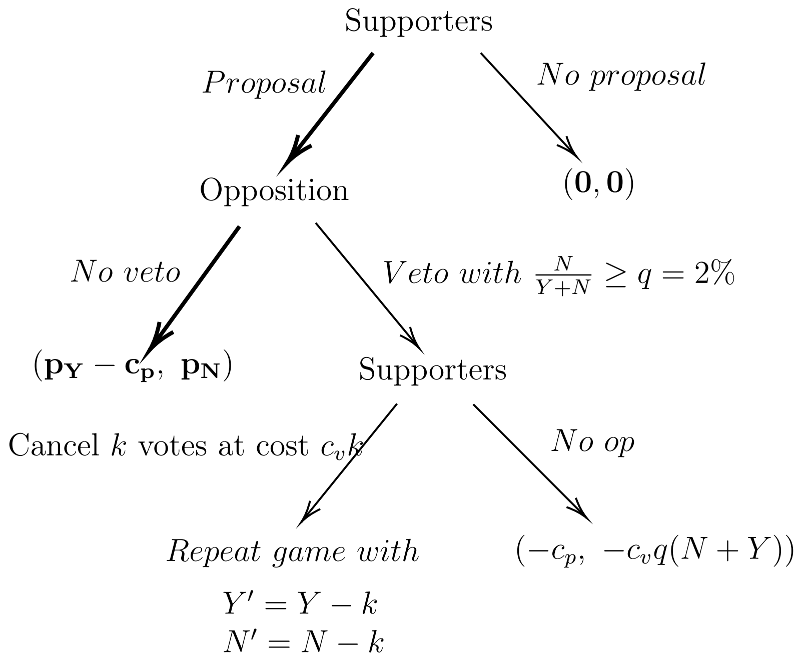

Supporters
Proposal
No proposal
Opposition
$(\mathbf{0}, \mathbf{0})$
No veto
Veto with $\frac{N}{Y+N} \geq q = 2\%$
$(\mathbf{p_Y} - \mathbf{c_p},\ \mathbf{p_N})$
Supporters
Cancel k votes at cost $c_v k$
No op
Repeat game with
$Y' = Y - k$
$N' = N - k$
$(-c_p,\ -c_v q(N+Y))$

Figure 2.6: Veto game. Offers the possibility to mutually reduce voting power and then repeat it.

governance system. This is done by looking at the two cases with and without majority support.

In the case with majority support, i.e. $Y > N$, and a proposal that was vetoed by the opposition, the supporters can mutually cancel N votes and repeat the proposal. This time, the opposition is left with $N' = N - N = 0$ votes and the supporters with $Y' = Y - N > 0$ votes, such that $N < q(N + Y)$, which means that the opposition has lost its veto power. The proposal passes.

In the case without majority support, i.e. $Y \leq N$, and a proposal that was vetoed by the opposition, the supporters can choose to mutually cancel up to Y votes. When they repeat the proposal, the opposition can still veto it as $N' = N - Y \geq 0$ and $Y' = Y - Y = 0$, such that $N' \geq q(N' + Y')$ still holds. The proposal does not pass.

Therefore, for both the veto game and the vote game with rational players, the proposal passes whenever $Y > N$. $\square$

2.4.4 Efficiency

This section takes the costs of the individual actions into account and argues without strict proof that passing a good proposal in the vote game is more expensive than in the veto game, making the veto game more efficient.

For the cost analysis, casting a vote or a veto is assumed to be associated with costs $c_v > 0$ per vote cast, so that casting y votes costs $c_v y$ and casting a veto costs $q(Y+N)c_v$. In line with established research, voter turnout y and n in simultaneous voting games with fully informed participants are assumed to be significant, i.e. in the same order of magnitude as Y or N, even if voting costs are high (Palfrey and Rosenthal, 1983).

Similarly, the costs for the mutual cancellation of k votes is assumed to be $c_v k$. The costs for making a proposal is assumed to be $c_p > 0$. Further, it is assumed that proposals can be repeated arbitrarily often.

Comparing the resulting payoffs for a passing proposal in the vote game $(p_Y - c_p - yc_v, p_N - nc_v)$ with that of the veto game $(p_Y - c_p, p_N)$, it is apparent that the veto game is potentially much more efficient as one can save the efforts of voting.

For these efficiency gains to materialize, it has to be assumed that proposals are mostly uncontroversial. This is the case if participants behave rationally and if there is an effective system in place to ensure that they are well-informed about Y and N. The two benchmark systems, Uniswap and Maker, run discussion forums to create a consensus and perform off-chain consultative votes in order to vet proposals before they are formally voted on in the expensive blockchain-based voting process. Another method to achieve a high degree of information about Y and N is to establish commonly accepted criteria for what makes a good proposal. Given such processes, we expect the Frankencoin's governance system to allow for a much larger number of good proposals passing that come from a broader spectrum of participants than the governance systems of the benchmark protocols described in section 2.1.1.

2.4.5 Attacks

In the context of decentralized autonomous organizations, it has to be assumed that passing a proposal is irreversible and therefore could potentially cause irreparable harm. This section formally shows that in the vote game, it is expensive to defend against such malicious proposals, while in the veto game, it is expensive to approve a good proposal against a malicious opposition. We argue that the latter is to be preferred.

Proposition 5 (Griefing with Votes)**.** [4]. *In a system based on majority votes, it is expensive for the majority to avert harmful proposals that come from a griefer, i.e. a player that is willing to pay c_p in order to cause disproportionate grief to others. Formally, given the payoffs $(-c_p - c_v n, -c_v y)$ from figure 2.5 for the averted proposal, there exists a strategy for the malicious supporters that forces the opposition to cast $n \geq Y$ votes even though the malicious supporters asymptotically do not need to vote at all, resulting in an asymmetric payoff close to $(-c_p, -c_v Y)$, allowing the supporters to cause disproportionate harm to the opposing majority.*

Proof. Given that there is a malicious proposal that would cause a lot of harm, i.e. $-p_N \gg c_v Y$, and that the supporters irrationally made despite knowing that $Y \leq N$, the opposing majority has no choice but to cast at least Y votes to rule out the possibility that the proposal passes.

The decision variable of the supporters is $y \leq Y$, denoting their actual number of votes cast. Likewise, $n \leq N$ is the decision variable of the opposition and denotes the number of votes cast against the proposal. Given that voting is costly, the players prefer not to vote if it does not make a difference. Both decision variables are chosen in a simultaneous game.

Under these circumstances, the supporters can cause disproportionate costs to the majority by adopting the following mixed strategy with an arbitrarily small $\epsilon > 0$:

$$y = \begin{cases} Y & \text{with probability } p = \frac{c_v Y}{-p_N} + \epsilon, \\ 0 & \text{otherwise} \end{cases}$$

The only rational answer for the opposition is to choose $n = Y$, the smallest number of votes that guarantees that the proposal is declined. This leads to the payoff of $-c_v Y$ for the opposing majority, whereas the supporting minority only has costs $c_p + pYc_v$, with

$$\lim_{p_N \to -\infty, \epsilon \to 0} c_p + pYc_v = c_p$$

[4]Griefing is a term borrowed from online gaming and denotes the intentional causing of harm for entertainment (Foo and Koivisto, 2004)

for proposals that cause a lot of harm p_N to the opposing majority. This results in the asymptotic payoff of $(-c_p, -c_v Y)$ for the supporters (the griefer) and the opposing majority. □

A majority-based system allows a malicious minority to launch griefing attacks in which both parties take a loss, but the loss of the good majority is much bigger than that of the attackers. Decentralized protocols usually discourage this type of griefing by requiring a minimum number of votes to make a proposal, for example 2.5% in the case of Uniswap. With this requirement, the attack requires substantial capital that is at risk of suffering from a capital loss as the value of the governance token can be expected to decline if the protocol is attacked. However, such a requirement also introduces an element of centralization and restricts the ability of making a proposal to only a handful of participants.

Note that voting on a public blockchain is not simultaneous as the votes become publicly visible as they happen. However, the participants could try to cast their votes immediately before the vote closes, making the game effectively simultaneous. Both benchmark systems, Uniswap and Maker, are in principle susceptible to such 'last minute attacks'. As a remedy, they both chose to impose high barriers for proposals to be made and to pass, making the governance process heavy.

Proposition 6 (Griefing with Vetos)**.** *In the veto-based system, it is expensive to push through a majority-supported proposal against a malicious opposition with power $N < 0.5(N + Y)$. More formally, an unknown minority only needs to spend Nc_v on casting vetoes whereas the cost for the supporting majority can grow infinitely high, depending on N.*

Proof. This proposition departs from the fully informed assumption. Otherwise, if the supporters knew the identity of the attackers, they could simply cancel their votes in advance and then pass the proposal at a total cost of $c_p + Nc_v$. However, if the griefers falsely identify themselves as supporters, their identities must be discovered incrementally during the veto game at high costs.

In each repetition of the game, the attackers cast a veto at cost $c_v q(N_i + Y_i)$, with Y_i and N_i denoting the voting power of each group at round i. In each round, the supporters cancel the votes of the newly revealed opposition members at cost $c_v q(N_i + Y_i)$ and repeat the proposal with costs c_p. These costs are determined by how often this attack can be repeated until the attackers do not have enough votes to cast a veto any more.

To formalize how many rounds that takes, consider the total number of votes $T_i = Y_i + N_i$ shrinks by the same factor in each round i, such that $T_i = (1-2q)^i T_0$. We are looking for the point in time k at which the opposition can cast the veto for the last time, which is at $N_k = qT_k$, implying $Y_k = (1-q)T_k$. Further using that both parties must have lost the same number of votes in the end, i.e. $N_k - N_0 = Y_k - Y_0$, one can solve for k:

$$k(Y_0, N_0) = \log_{1-q} \frac{Y_0 - N_0}{(Y_0 + N_0)(1-2q)}$$

Notably:

$$\lim_{N_0 \to Y_0} k(Y_0, N_0) = \infty$$

So in case of a large unknown opposition that almost has a majority, the griefing potential is unbounded as each of the potentially infinite repetitions comes with the costs c_p for renewing the proposal. $\square$

However, for small N_0, the number of rounds needed is approximately $\frac{N_0}{q}$, which implies that increasing the quorum q needed to cast a veto can reduce the vulnerability of the system. At the same time, having a low q makes it more risky to launch controversial proposals with an unknown opposition, making the system more conservative overall.

2.5 Fundamental Value

We now address the conditions under which the *peg to the Swiss franc* or any other reference currency of choice should hold. The fundamental value of the Frankencoin comes from a combination of three elements: first, it is necessary to have enough collateral such that eventually, each ZCHF can be sold for at least one Swiss franc to a minter who wants to get their collateral back. Second, the interest rate of the system (r_d as defined in section 2.3.3) should be managed such that given a non-zero value of the stablecoin at some point in the future translates into a current value that matches the reference currency. Third, one needs some assurance for the minters to ensure that they cannot be made subject to a short-squeeze. These three factors are addressed in the following three sections.

2.5.1 Sufficient Backing

On the supply side of the Frankencoin, pool share holders should not allow any mintings that create a risk of the value of the supplied collateral falling below the nominal value of the Frankencoins in circulation. Assuming that each Frankencoin is always backed with at least one Swiss franc worth of collateral, the Frankencoin is fundamentally sound and one can assume that the minters are willing to pay at least one Swiss franc per Frankencoin at the point in time when they want to get their collateral back again. This is taken care of by the collateralized minting mechanism discussed in section 2.2.

2.5.2 Comparable Yield

On the demand side, we approach the valuation of the Frankencoin from the perspective of a *perpetual bond*. A perpetual bond, or consol, is a bond with coupon payments but no redemption date (Jorion et al., 2010). We price this perpetual along the lines of (Jarrow and Turnbull, 2000), by discounting the interest payments on a risky term structure. Let's assume that interest payments happen at discrete time-steps $0, ..., \infty$ and we have corresponding risky rates of the term-structure so that the date-0 value of a promised Swiss franc at time t of a risky Franc promise is equal to $\exp(-r_t t)$. Let the constant coupon rate per Frankencoin be c. Now, the value of the perpetual can be written as

$$v(0) = \sum_{t=0}^{\infty} c e^{-r_t t} \tag{2.2}$$

$$= \sum_{t=0}^{\infty} c e^{-yt} \tag{2.3}$$

$$= \frac{c}{1-e^{-y}}, \tag{2.4}$$

where the second line replaces the time-specific discount rates by a yield, and the last line is an application of geometric series. For the value to be at par, $v(0) = 1$, we have to choose the coupon rate accordingly: $c = 1 - e^{-y}$. Hence, if the interest earned from contributing ZCHF are in line with discounting, the present value of one ZCHF is equal to one Swiss franc.

The risky term-structure corresponds to the Swiss franc risk-free term-structure plus a spread that compensates the investor for the risks. Whenever the risk-free term-structure, or the Swiss franc risk changes, c has to be adapted for the value $v(0)$ to be equal to one. In the Frankencoin system, the yield is implicitly set by the pool share holders as they can veto minters that do not yield the right risk-adjusted return for the system. In effect, the pool share holders act as an oracle, but for long term interest rates instead of short term prices.

For this to work, minters must be required to pay an interest on their open positions and positions should be limited in time. The shorter the average term of the open positions is, the faster it is possible to push the yield that can be earned from buying pool shares to the correct c. Generally, the system should seek to increase the yield if there is too little demand for holding ZCHF and vice versa.

The comparison with the consol shows nicely that it is not necessary for a stablecoin to be directly convertible to the reference currency in order to have the same fundamental value. It suffices if the coin offers the same expected return. Thereby, the problem is reduced to the ability to pay out an equivalent interest under the assumption that there are market makers that recognize the long-term value and engage in arbitrage trading in order to avert short-term deviations from the peg.

2.5.3 Price Ceiling

While it is generally not possible to exchange Frankencoins directly into collateral provided by the minters, the minters will have to buy back their minted Frankencoins before they can get their collateral back. Here, the minters face the risk of a short squeeze. By minting and selling Frankencoins, they are short ZCHF and might be forced to pay more than one Swiss franc per Frankencoin to unlock their collateral. So while those holding the stablecoin face the risk of it falling below the peg, minters face the risk of the Frankencoin departing upwards from the peg.

In the proposed setup, we start with a very simple mechanism to avert the risk of an overvaluation: we provide a bridge contract that allows holders of other Swiss franc based stablecoins to convert them 1:1 into Frankencoins. As long as such bridges exist, minters can be confident that they do not need to overpay for the unlocking of their collateral. However, while relying on other stablecoins can help in practice, it is not desirable to depend on external systems.

In the absence of stablecoin bridges, minters have to trust the system to always allow the minting of new Frankencoins at competitive terms, such that a short-squeeze can be averted by simply minting additional Frankencoins and repaying the open position with those. In effect, the system relies on good governance on both the supply and demand side. On the supply side, the system must allow economically sensible mint contracts and disallow irresponsible ones. On the demand side, the system must ensure that the risk-adjusted interest rate tracks that of the Swiss franc.

2.5.4 Long-term Incentives

Like many blockchain-based systems, the Frankencoin relies on the rationality of its participants. There is no technical barrier for the participants to collectively mismanage the Frankencoin, for example by letting a buggy minter contract pass. It is in the economic interest of all involved parties to look after the system and to make it work as expected. In particular, pool share holders have to be aware that they would destroy the long-term value of the system by letting the stablecoin deport too far from the peg. Since their reserve is at stake and they are the last who could cash out in case of a crash due to a loss of trust, pool share holders have a strong incentive to not let the price fall. At the same time, they also have a strong incentive to ensure is a wide range of available mint contracts such that the risk of a short-squeeze for minters is under control.

The reason why the system has such a strong interest in making sure the Frankencoin tracks the value of the Swiss franc is that it enables them to earn net interest income to the extent that the Frankencoin is used as a transactional currency. For example, if the open market interest rate is 3% without any margin between borrowers and lenders, the equity holders collectively earn around 9% on the deployed capital in equilibrium as outlined in section 2.3.3. This is only possible if the Frankencoin is valuable enough as a transactional currency such that two thirds of the holders do not care about missing out on potential interest. And the transactional value is highest when the Frankencoin reliably tracks the value of the reference currency.

2.6 Implementation

This section briefly touches on the software architecture of the Frankencoin as shown in figure 2.7. In line with the insights presented in *The Code is the Model*, the source code should be considered the ultimate specification of the system, with the text and mathematical parts of this chapter serving to document, analyze, and to generally present the system in a commonly accessible way (Meisser, 2017).

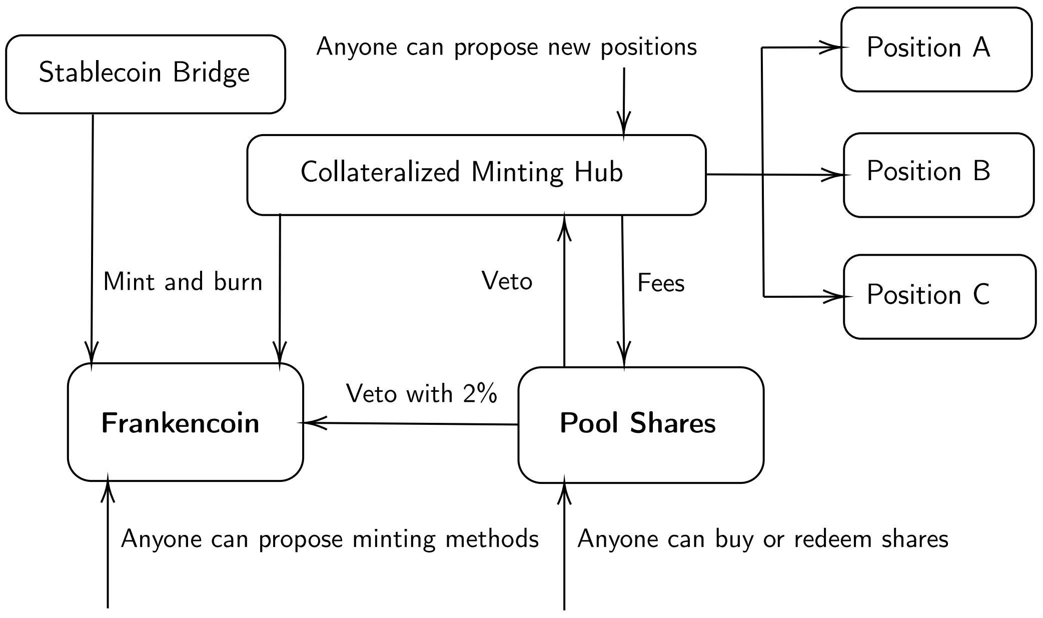

Stablecoin Bridge
Anyone can propose new positions
Position A
Collateralized Minting Hub
Position B
Mint and burn
Veto
Fees
Position C
Veto with 2%
Frankencoin
Pool Shares
Anyone can propose minting methods
Anyone can buy or redeem shares

Figure 2.7: Architecture. There are two ERC-20 tokens, the Frankencoin (ZCHF) and the Frankencoin Pool Shares (FPS), both with extra functionality to allow new minters to be proposed subject to the veto-based governance process. The diagram shows two types of minters, the stablecoin bridge and the minting hub. The latter contains all functionality needed to initiate new collateralized minting positions.

The richly commented source code can be found in the end of the chapter starting with appendix 2.D and ending with appendix 2.J. The Frankencoin smart contracts are written in Solidity, a programming language developed for the Ethereum system and also supported by other blockchains that are compatible with the Ethereum Virtual Machine (EVM). The code listed in the appendices does not include standard contracts such as ERC-20 and Ownable (Vogelsteller and Buterin, 2015). Furthermore, interface declarations are excluded to avoid redundancy. The full source code can be found on github.com/Frankencoin-ZCHF/FrankenCoin.

The source code underwent three security audits over the course of the year, by Scherer (2023), code4rena (2023), and Mackinga, Ulqinaku, et al. (2023). Besides uncovering a number of technical vulnerabilities, the audits led us to change the auction process from a traditional auction with competing bids to a Dutch auction. In a Dutch auction, the price starts high and is continuously reduced until a buyer appears. This type of auction is less susceptible to last-minute manipulations and has requires less information to be stored on chain.

2.7 Risk Analysis

We start by analyzing the risk of a capital loss section 2.7.1, leading us to a discussion of meaningful capital requirements in section 2.7.2, and concluding with a proposal for risk-based capital requirements inspired by Basel iii rules in section 2.7.3.

We find that with the chosen parameter $h = 0.1$, implying a loan-to-value ratio of $l = \frac{1}{1+h} \approx 0.91$, using Bitcoin price data and an auction duration of 24 hours, the system earns a significant net profit from the simulated liquidations and therefore should not need additional risk compensation in the form of an increased interest or otherwise higher fees. These results serve as a foundation to calibrate risk-based capital requirements, where we find when applying Basel iii rules in a technology-neutral way, equity ratios in the low single digit percentage point range would suffice and note that the Frankencoin system has a generous risk buffer in its current design.

2.7.1 Risk from Collateralized Minting

The Frankencoin system is at risk of losing equity capital if a liquidation ends at a price that does not suffice to cover the outstanding stablecoin balance and the challenger reward. To evaluate that risk, we abandon the earlier assumption that the price is constant during the auction. It is also noted that a rational challenger will not immediately start a challenge when the market price p hits the liquidation price p_T, but at a price level $p_0 < p_T$ slightly below the trigger price that depends on the opportunity costs of having the collateral locked up and the probability of the challenge succeeding. Based on these considerations, simulations are performed to evaluate the risk of a loss depending on the price level at which the challenge was started using historical Bitcoin price data.

In deviation from the previous section 2.2.3, the bidding process is assumed to be a normal auction that runs for a fixed amount of time τ or ends immediately when the highest bid reaches the liquidation price, i.e. $p_B = p_T$. Also, the challenger reward is not based on the bid as before (i.e. kp_BC_C), but based on the minted amount of Frankencoins (i.e. klp_TC_C). These deviations should not make a notable difference in the simulation results, so it was decided against redoing them after the changes were introduced. As before, we note that the rational bids imply prices matching (exogenous) market prices. We do not account for slippage incurred when selling the collateral in exogenous markets so that we can use Bitcoin market prices for the exercise at hand.

We define p_0 as the exogenous market price at the time the challenge is started and further define $h = \frac{1}{l} - 1$ and $h' \leq h$ the level of overcollateralization (or undercollateralization if $h' < 0$) at the point in time when the challenge is initiated. Consequently:

$$p_0 = (1 + h')lp_T \tag{2.5}$$

From Section 2.2.3 we have the *condition for liquidation*:

$$p_B < (1 + h)lp_T, \tag{2.6}$$

to which we add the time dimension now. We consider the asset return $\tilde{r}_t$ over the liquidation horizon τ. The challenge is averted earlier after t' if during the liquidation period a bid at the liquidation threshold is made. We can now express the condition for liquidation as a function of the asset return, and the values h and h':

$$e^{\tilde{r}_t} < \frac{1 + h}{1 + h'} \quad \forall 0 \leq t \leq \tau. \tag{2.7}$$

This is another way of expressing that the challenge is successful if at no point in time t the price reaches p_T.

Proof. By definition, given the return r_τ we can express the price at the end of the auction as $p_\tau = exp(r_\tau)p_0$. Using $p_B = p_\tau$ in the case of the successful auction, one can reformulate inequality 2.6 as follows:

$$p_B = p_\tau = exp(r_\tau)p_0 = exp(r_\tau)(1 + h')lp_T < (1 + h)lp_T$$

which trivially leads to inequality 2.7 by dividing by $(1 + h')lp_T$. □

To avoid the risk of the market price declining below an already made bid, a rational bidder will wait with the placement of the bid until near the end of the auction. We can therefore assume that successful challenges lead to a liquidation at price p_τ even if the market price was temporarily higher during the auction.

The profit $\tilde{P}$ for the system is zero in case the challenge is averted, positive in case of a liquidation at a price $p_\tau \geq (1+k)lp_T$, negative if the price is below, and zero if the challenge is averted and no liquidation takes place. We define a dummy variable D that is exactly 1 in case a challenge is successful and 0 otherwise:

$$D = \mathbf{1}_{e^{\tilde{r}_t} < \frac{1+h}{1+h'}\ \forall\, 0 \leq t \leq \tau} \tag{2.8}$$

We can now express the profit $\tilde{P}$ per unit of the challenged amount as the following random variable:

$$\tilde{P} = \left[(1+h')e^{\tilde{r}_\tau} - (1+k)\right] D \tag{2.9}$$

Note that if the challenge starts at $h' \geq h$, the indicator function is always $D = 0$ as the challenge is immediately ended again. We now determine the value $\tilde{P}$ to decide whether the system needs to be compensated by a minting fee.

To tackle the valuation of $\tilde{P}$, we resort to the arbitrage free pricing principle, which states that the value of a contingent claim is given by its discounted expected value under the risk-neutral probability measure (Björk, 2009). We assume that the risk-free rate used for discounting is equal to zero, which we consider adequate especially since the period τ is very short. Let $f_\tau(x)$ be the density function for the return distribution over the period τ. Now, we can value $\tilde{P}$ conditional on the starting level of liquidation h' as follows

$$\mathbb{E}^{\mathbb{Q}}_{\tau}\left[\tilde{P}|h'\right] = D \int_{-\infty}^{\log \frac{1+h}{1+h'}} \left[(1+h')e^{x} - (1+k)\right] f_\tau(x)dx \tag{2.10}$$

where the subscript τ emphasizes that the distribution depends on the liquidation period.

We now discuss h'. If we knew the distribution of starting levels h', we could integrate out h' to arrive at $\mathbb{E}^{\mathbb{Q}}_{\tau}\left[\tilde{P}\right]$. As established, challengers trade off the probability of collecting the reward given they issue a challenge versus the probability of not being able to be front-run by other challengers. To investigate this trade off, we assume that challengers issue a challenge when the collateral value reaches a level for which there is a given probability α that they receive the challenger reward. We then investigate what minting fees each h' (or equivalent, each α) implies.

Equation (2.10) uses the risk-neutral probability measure, often referred to as $\mathbb{Q}$, rather than the objective measure $\mathbb{P}$. In practice, parameters for the measure $\mathbb{P}$ are extracted directly from market data (e.g., sample volatilities and expected returns), whereas parameters for the measure $\mathbb{Q}$ have to be extracted from option data under the same model assumptions (e.g., option implied volatilities). With risk averse investors, the $\mathbb{Q}$-measure puts more weight on adverse market events, see for example Breeden and Litzenberger (1978), leading to a higher risk-neutral price of $\tilde{P}$, compared to the value obtained when integrating Equation (2.10) under the objective measure. We therefore proceed by calibrating a probability distribution to observed market data and use this as a lower bound for the price of $\tilde{P}$, or, equivalently the minting fee should be at least equal to the negative value of $\tilde{P}$ under the measure $\mathbb{P}$:

$$\Theta_F^{(i)} \geq -\mathbb{E}_\tau^{\mathbb{P}}\left[\tilde{P}\right], \tag{2.11}$$

where we add the negative sign because $\tilde{P}$ is a profit. For brevity, we omit the superscript $\mathbb{P}$ in the sequel.

Challenge success probability

We are now ready to calibrate the parameters. To do so, we evaluate the probability of liquidation for different challenger levels h'. We then locate reasonable challenger levels and investigate the lower bound of the minting fee, $\Theta_F^{(i)}$, for the given level of h'. In Appendix 2.C we describe the BTCCHF candle-data that we use to calibrate the fees. We use 24h candle data (Open, Low, High, Close) and use the bootstrap method introduced by Efron (1992) for estimation.

With candle data, we can readily determine whether for a given period, the challenge started at the beginning of the candle would have been averted, $D = 0$, or succeeded $D = 1$, by

$$\hat{D}_\tau = \mathbf{1}_{\left\{\frac{P_H}{P_O} < \frac{1+h}{1+h'}\right\}}, \tag{2.12}$$

where P_O is the open price, P_H the high price over period τ (without indexing the candle for brevity), which follows directly from definition of D in Equation (2.8). For the bootstrap, we define the following variables. Let N be the number of candle observations, B the number of bootstrap replications, and $\hat{\mathbf{D}}_b = \{\hat{D}_\tau^{(b,1)}, ..., \hat{D}_\tau^{(b,N)}\}$ the b^{th} bootstrap replication for a period-τ dummy that equals one if the challenge was not averted. The bootstrap estimate for the probability of liquidation follows directly from Equation (2.12):

$$\hat{\mathbb{E}}\left[D|h'\right] = \frac{1}{B}\sum_{j=0}^{B-1}\frac{1}{N}\sum_{n=1}^{N}\hat{D}_\tau^{(j,n)}. \tag{2.13}$$

For comparison, we also evaluate the probability of liquidation conditional on h', assuming the challenge cannot be averted prior to time τ. For this assumption we use both, a bootstrap evaluation (replacing P_H/P_O by P_C/P_O, with P_C equal to the close price, in Equation (2.12)) and a closed-form evaluation assuming the returns are normally distributed:

$$\hat{\mathbb{E}}\left[\mathbf{1}_{\{\tilde{r}_\tau < \log\frac{1+h}{1+h'}\}}\middle| h'\right] = \Phi\left(\frac{\log(1+h) - \log(1+h') - \mu_\tau}{\sigma_\tau}\right), \tag{2.14}$$

where μ_τ and σ_τ are location and scale parameters of the normal distribution, and $\Phi(\cdot)$ represents the standard normal cumulative distribution function. We use the sample mean and standard deviation of the observed returns to estimate μ_τ and σ_τ respectively. Figure 2.8 shows the results. We see that allowing the challenge to be averted early significantly reduces the probability that a challenge is successful when it is issued at a price close to the liquidation threshold. At $(1 + h')$ about 2% below the liquidation threshold $(1 + h)$, there is a fifty percent chance of the challenge being successful.

The normal distribution overestimates the probability of liquidation for values h' above the liquidation threshold h, and underestimates the liquidation probability for value of $h' < h$ compared to the bootstrap result. We use the variance resulting from the bootstrap replications to estimate confidence intervals.

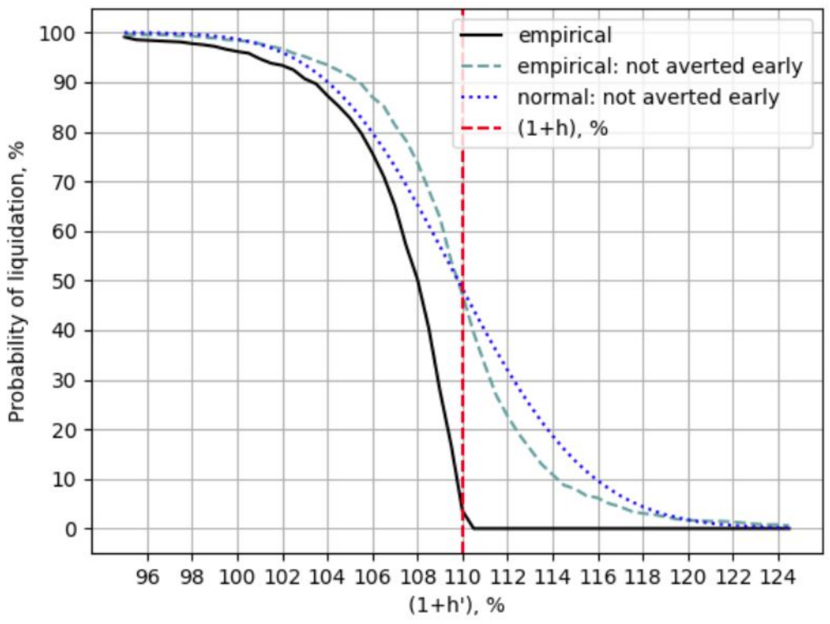


Figure 2.8: Probability of liquidation given h'. This figure plots the probability of the challenge ending in a liquidation, when the challenge is started at a level h', with the liquidation level at $h = 0.10$ and a challenge duration of 24 hours (solid black line). The probabilities are estimated via bootstrap ($B = 5,000$ samples). Confidence intervals at the 1%-level remain below 0.05% around the point estimates for each h'. For comparison, we hypothetically assume the challenges cannot be averted before the end of the liquidation horizon (dashed and dotted curves). We compare the bootstrap estimate (dashed line termed "empirical: not averted early"), with the theoretical probability using normally distributed returns (with sample mean and standard-deviation).

Minting fee valuation

Finally, we use use Equation (2.9) to arrive at our bootstrap point estimate for the value of the liquidation profit P.

Let $\hat{\mathbf{r}}_b = \{\hat{r}_\tau^{(b,1)}, ..., \hat{r}_\tau^{(b,N)}\}$ the b^{th} bootstrap replication for a period-τ return, now

$$\hat{\mathbb{E}}_\tau \left[\tilde{P}|h'\right] = \frac{1}{BN} \sum_{j=0}^{B-1} \sum_{n=1}^{N} \hat{D}_\tau^{(j,n)} \left[(1+h')e^{\hat{r}_\tau^{(j,n)}} - (1+k)\right]. \tag{2.15}$$

We use $B = 10,000$ bootstrap replications and calculate confidence intervals at the 1%-level by applying the central limit theorem.[5] The confidence interval result in ranges in the 0.01%-area. Figure 2.9 shows the results. At low challenge levels, the contributors have to burn ZCHF and there is a loss for them. At very high challenge levels, often the challenge does not result in a liquidation and hence there is no gain or loss for contributors. The system makes the most profit for challenges that start at a level of about $1 + h' = 107\%$.

In Figure 2.9 we see that the system does not lose on average if the challenge starts above $1 + h' = 102\%$. From Figure 2.8 we learn that starting the challenge process at $1 + h' = 102\%$, there is about a 90% probability that the challenge ends in a liquidation, hence the challenge is expected to start above the 102% level. We conclude that the minters do not need to be charged a risk premium to compensate the system, unless risk-aversion would be very high.

2.7.2 Capital Requirements

Banking regulation prescribes three types of capital requirements for which we find a meaningful analogue in the Frankencoin system, see for example the Basel III banking regulation (Basel Committee on Banking Supervision, 2010) or the Dodd-Frank Act (Acharya et al., 2010).

1. *Risk-based capital requirements.* These reserve requirements are based on the sum of the reserve requirements of each individual position, whereas each position can have its own risk weight. The equivalent to that in the Frankencoin

[5]Having a vector of B bootstrap estimates $\mathbf{x}$, we report the point estimate as the mean of $\mathbf{x}$, and the error at the a-level as $z_{1-a}\sqrt{V[\mathbf{x}]/B}$ with z_{1-a} the standard normal quantile.

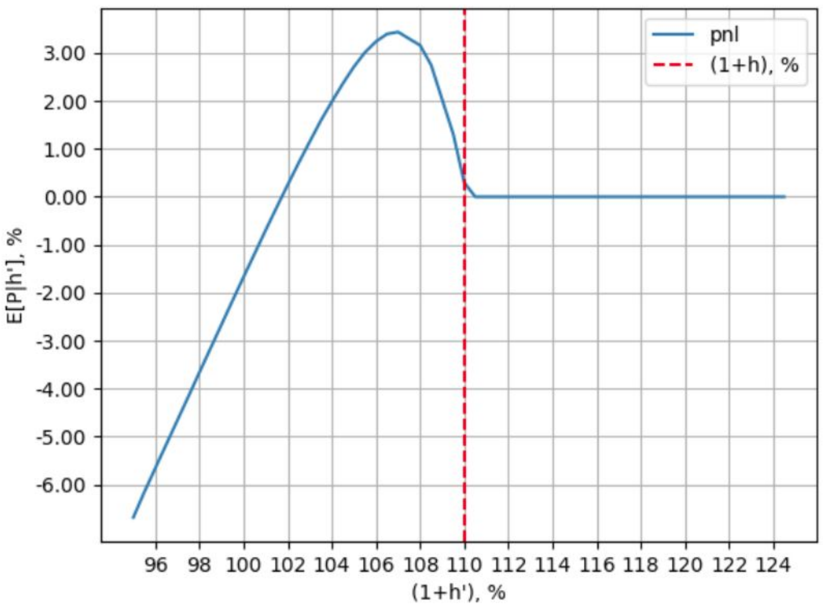


Figure 2.9: Expected system profit: We plot the expected profit (or loss) to the system depending on the level h' at which the challenge was started. Negative values observed for approximately $1+h' < 102\%$ correspond to a loss for the system. Challengers are likely to start the challenge at a level above $h' = 0.02$ which corresponds to a liquidation probability of over 90% as we see in Figure 2.8. Therefore we conclude that with these parameters, we do not need to compensate the system with a minting fee, unless there is a high risk-aversion.

system is the requirement to contribute to the minters reserve, which might differ for each position based on its riskiness.

2. *Leverage limits* restrict the overall amount of leverage in the balance sheet. The balance sheet of Frankencoin is presented in Appendix 2.A. Leverage limits are largely model-free and are therefore a safeguard against model risk (e.g., model risk arising from the risk-models used to calibrate plugin-specific capital requirements). Further, a leverage limit provides us with a global (i.e. Frankencoin-system-wide) capital limit. In the Frankencoin system, the global leverage limit is not directly enforced but economic incentives are designed such that in equilibrium, the system should exhibit a robust leverage ratio. On top of that, the collective minters reserve provides an additional layer of protection that is comparable to the leverage limit.

3. *Concentration limits.* If the collateral of Frankencoin was mainly exposed to the price of one asset, Frankencoin could more easily collapse following a large price drop of that asset, compared to a more diversified collateral in the system. The concentration limit aims at limiting the exposure to a single asset or a group of related assets.

The main difference between traditional Lombard loans and the Frankencoin's collateralized minting positions is the treatment of the proceeds in case of a liquidation. When a traditional Lombard loan is under-collateralized and the margin call not answered, the lender only sells as much collateral as needed to restore the loan-to-value ratio. Also, excess proceeds from the sale are returned to the owner of the collateral. In contrast, the Frankencoin system liquidates the whole positions as soon as the price reaches the trigger level and collects excess proceeds as profits. The range between the loan-to-value level and the liquidation level can be considered as conditional capital contribution by the borrower and for capital requirements considerations, this conditional capital has equity-like risk-absorbing properties. So in effect, both the equity and the minters reserve add to the risk-absorbing capacity of the system and should be taken into account when calculating the equivalent to the leverage ratio for banks. With regards to the minters reserve, the system resembles clearing houses, where each clearing member is required to contribute to the guaranty fund based on the risk of their position.

2.7.3 Risk-based Capital Requirements

Lombard loans are loans extended by banks to their customers, secured by the customers' securities that are held in bank custody. In the Lombard loan agreement, bank and customer agree on the terms of the loan, including that additional assets have to be provided in case the value of the collateral falls below a margin call level. The bank has the right to sell the pledged securities, if the value falls below an agreed level. In the Frankencoin system, there is no agreement beyond what is defined in the smart contract, and hence there will not be any bankruptcy litigation. As a consequence, there is a loss to the system that has to be covered with Frankencoin reserves, as soon as the collateral drops below the loan amount. Not surprisingly, this difference renders the Basel iii capital requirements for collateralized loans inadequate for the Frankencoin system. We therefore propose a more conservative approach to capital reserves.

Each mint plugin defines the required reserves of ZCHF to be held against the issued volume of ZCHF. We now assess how to specifically determine the risk-based capital requirements for our illustrative setup. We do not require risk-based capital reserves for the bridge plugins, so we focus entirely on the collateralized mint plugin, collateralized in Bitcoin.

The assessment we perform now differs from the one we made to determine the minimal fee in two main aspects, (1) we want to estimate a loss for the whole collateralized mint plugin not only for an individual position, and (2) we are not looking for an insurance valuation but for the capital required. The latter point implies that we do not need any assumption on risk-neutral measures but we can directly work with the observed data and use real world probabilities.

We start by applying the Basel iii rules for risk based capital.

From Basel Rules to Frankencoin Capital Requirements

The Basel Committee of Banking Supervision defines capital rules for collateralized loans in the Basel iii regulatory framework. In view of the authors, the collateralized mint plugin best meets the conditions to be considered a “repo-style transaction", see 22.66 in Basel Committee on Banking Supervision (2019a), henceforth [CRE22]. In this framework, the bank first weights assets according to the risk to obtain the risk-weighted assets (RWA). The bank is then required to hold a certain amount of capital, for example Common Equity Tier 1 must be at least 4.5% of RWA and total capital must be at least 8% of RWA, for details see Basel Committee on Banking Supervision (2019b). For collateralized loans, the rules are detailed in [CRE22]. Banks have two options: *"Banks may opt for either the simple approach, which substitutes the risk weighting of the collateral for the risk weighting of the counterparty for the collateralized portion of the exposure (generally subject to a 20% floor), or for the comprehensive approach, which allows a more precise offset of collateral against exposures, by effectively reducing the exposure amount by the value ascribed to the collateral"*, see 22.12 in [CRE22].

We focus on the comprehensive approach. First, the bank determines the exposure amount after risk mitigation, E^*, which depends on the loan and the collateral. Both are subject to a haircut, see 22.40 in [CRE22]:

$$E^* = \max[0, E(1 + H_e) - C(1 - H_c - H_{fx}), \tag{2.16}$$

where E^* is the exposure amount after risk mitigation, E the current value of the exposure (the loan), H_e haircut appropriate to the exposure, C the current value of the collateral received, H_c the haircut appropriate to the collateral, H_{fx} the haircut appropriate for currency mismatch between the collateral and exposure. The exposure amount E^* is to be multiplied by the risk weight of the counterparty to obtain the risk weighted asset amount for the collateralized loan. Collateral haircuts are either determined based on the type of collateral (e.g., 25% for 'other equity' and a holding period of 10 days). The exposure is then multiplied by the risk weight of the counterparty to obtain RWA. If the counterparty is not rated, a weight of 1.5 is applied. Figure 2.10 summarizes this approach. In Appendix 2.B we calculate the Basel iii capital requirements using the comprehensive approach and using the approach with own haircut estimates based on a 99%-VaR as per Basel iii. Table 2.6 shows the results: per Basel, the required capital reserve results to only about 1% of outstanding loans, even when assuming a loan-to-value ratio of $1/(1 + h) = 1/1.1$ (which is always below current exposure assuming liquidations are effective).

To render the Frankencoin system resilient, the capital requirements must not assume that litigation is possible, hence the whole exposure after risk mitigation should be covered by capital reserves, as opposed to 12% per Basel iii. Second, instead of applying the haircut on the current exposure value, $(1 + h^*)$ as per Basel iii, we apply the haircut on a conservative liquidation level $(1+h')$. This is illustrated in Figure 2.10. To calculate the haircut, we estimate an empirical VaR at the 99%-

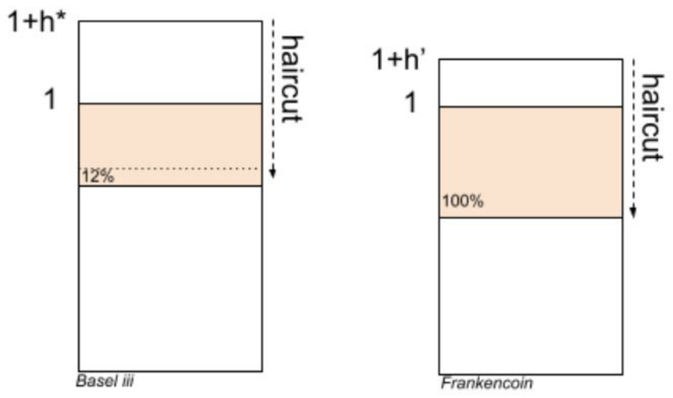


Figure 2.10: Basel iii vs own approach. Basel iii subtracts a haircut from the current collateralization value $(1 + h^*)$ to obtain the exposure after risk mitigation (shaded area). This quantity is multiplied by 1.5 for unrated counterparties. Total capital is to be 8% of this. However, in the Frankencoin system there is no loan agreement beyond the smart contract, and hence no litigation. We therefore propose a more strict approach, in which (1) the calculation of the haircut starts at a conservative liquidation level $(1 + h')$, (2) the entire exposure after risk mitigation is to be covered, as opposed to $1.5 \cdot 8\% = 12\%$.

level (a VaR level in line with Basel iii). We call this the Frankencoin Proposal in Table 2.9. Second, we look at the Basel iii requirements for collateralized lending

Method	Capital
Frankencoin Proposal (empirical VaR)	10%
RWA based, Basel iii Comprehensive	1.1%
RWA based, Basel iii Comprehensive (own haircuts)	1.3%

Table 2.9: Risk-based capital requirements. Results of capital reserves required using different approaches. For the setup at hand, we recommend that the system should hold 10% of the outstanding loans as Frankencoin reserves.

Capital Estimation per Frankencoin Proposal

We construct empirical samples of loss data by applying the 24h log-returns to the loss function given by Equation (2.9), which we repeat here:

$$\tilde{P} = \left[(1 + h')e^{\tilde{r}_\tau} - (1 + k)\right] D,$$

and we multiply the samples by -1 to have a positive number for a loss. Figure 2.11 gives an overview of the loss data for different challenger levels h' via boxplots. We see that the median level of losses are benign, even for very low challenger levels of $h' = -1\%$ (meaning the liquidation is only started when the collateral has a value of 99% of the loan notional, although the position would liquidate if the collateral is at 110% of the loan notional). However, we also see that there are high losses for all levels of h' depicted, driven by two extreme returns in the data, where the loss exceeds 20% of the loan notional. These losses are at a loan level and apply if the loan challenge starts at a collateral coverage ratio of $(1 + h')$. On a plugin-wide level, we would only lose the same percentage of the outstanding loan notional, if all loans started at $(1 + h')$ simultaneously. Realistically, collateral coverage ratios are more spread out and hence, the loan-level losses observed in Figure 2.11 are upper bounds to the plugin-wide loss.

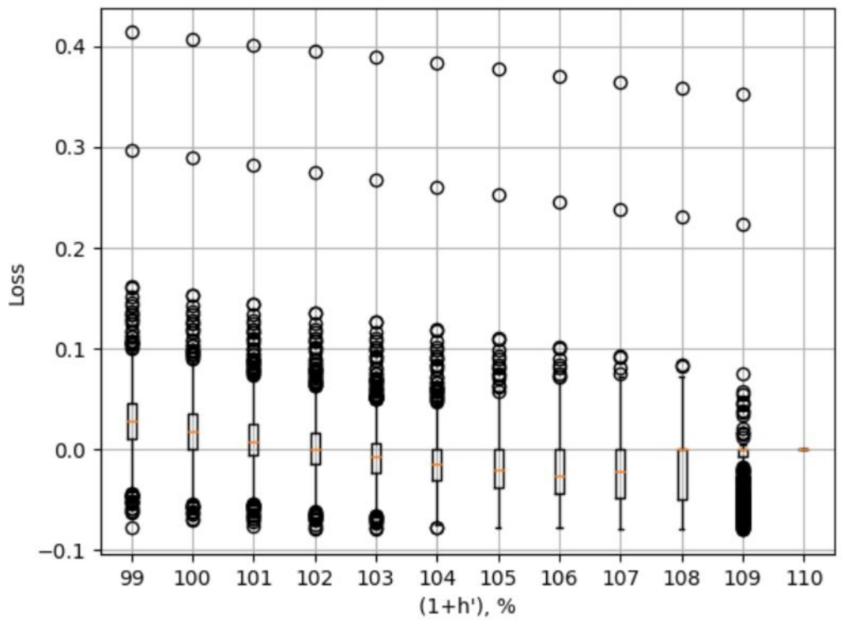


Figure 2.11: Empirical loss boxplot. This figure shows standard boxplots for different challenger levels h', given that a liquidation is triggered when the challenge results in a collateral value of below 110% of the loan, and a challenger fee of $k = 2\%$. The vertical bar depicts the median, the boxes reach from the first quartile to the third quartile, the whiskers extend to the max/min or at most 1.5 times the range of the box, circles are plotted outside the span of the whiskers. For values h' closer to $h = 0.10$, the loss more often ends in a gain (negative number). At $h = 0.10$ the challenge is averted. Losses stay on average benign but we also observe a few very high losses that exceed 20% of the loan notional due to two very negative returns (-49%, -31%).

Table 2.10 presents the 99-percentiles for daily losses (using the loss empirical loss distribution seen in Figure 2.11, given the challenge start level h'. A conservative starting level is $h' = 2\%$ as we have seen in the previous chapter. The associated 10.73% loss at the 99%-level would assume that all loans have an equally low collateralization of h'. To conclude, using our proposed approach that deviates from the Basel iii rules towards the conservative side, we propose to set the capital requirement to 10% of the outstanding loans.

h'	VaR99, %
-0.01	13.42
0.00	12.52
0.01	11.63
0.02	10.73
0.03	9.84
0.04	8.94
0.05	8.05
0.06	7.15
0.07	6.26
0.08	5.36
0.09	3.77
0.10	0.00

Table 2.10: Empirical VaR. This table shows the empirical VaR in percent of the outstanding loan amount given the challenge starting level h'.

2.7.4 Resulting Leverage Ratio

We have observed in section 2.3.3 that in equilibrium, we can expect equity capital to be about a third of the effectively borrowed capital. Furthermore, there is a minters reserve that can be used to cover losses once the equity holders have been wiped out. Assuming that the average minters reserve requirement is about 20% of the borrowed capital, we arrive at an exceptionally conservative leverage ratio of above 50%! However, taking into account that it is further possible to mint stablecoins through stablecoin bridges as depicted in the balance sheet of appendix 2.A, this leverage ratio can be lower to the extent that the bridges are used. To guard against the imported risks from bridged stablecoins, the bridges have built-in limits for the amount of Frankencoins that can be minted by them. Given this conservative approach, the main risk for the Frankencoin system does not seem to stem from the under-collateralization of individual positions, but from a potential lack of competitiveness with other protocols that are willing to take more risks.

2.8 Conclusion

Blockchain technology opens the possibility to create a new kind of monetary institutions that are governed in a decentralized way. Unlike bank loans that are issued by a centralized entity, the presented Frankencoin can be minted by the borrowers themselves, given that they provide suitable collateral, and the system has sufficient capital in its reserve. In the presence of a central bank digital currency (CBDC), see for example Chaum, Grothoff, and Moser (2021) or Brunnermeier and Niepelt (2019), the presented Frankencoin could come with a bridge to that CBDC, in which case the Frankencoin system could be seen as a fractional reserve multiplier just like the established banks are for traditional central bank money. With that perspective, it is no surprise that risk mitigation strategies from the current banking regulation can serve as a starting point to risk mitigation in the Frankencoin system.

Appendix

2.A Frankencoin Balance Sheet

Figure 2.A.1 presents a stylized balance sheet view of the Frankencoin system. When central banks print and issue currency, the outstanding amounts appear on the liability side of the balance sheet. Similarly, the minted Frankencoins also appear on the same side of the balance sheet. To see how they end up there, one first needs to consider the balance sheet of an individual minter. When a minter mints new Frankencoins, the freshly minted coins appear on the asset side of the minter's balance and at the same time, a repayment obligation is created on the liabilities side. For the Frankencoin system, the opposite happens: the minter's repayment obligation appears on the asset side and the minted ZCHF among the liabilities.

One should note that the total balance sheet of the Frankencoin system is larger than the total number of Frankencoins in circulation. That is because the same Frankencoin can appear multiple times. If a minter sells his ZCHF and the buyer uses them to buy Frankencoin Pool Shares, these Frankencoins appear a second time on the balance sheet, this time as reserves with a corresponding increase in equity on the passive side. Similarly, when for example minting 100 ZCHF against a collateral with a loan-to-value ratio $l = 0.8$, it is not only the total supply and the minter repayment obligations that increase by 100 ZCHF each, but also the reserve and the minter reserve go up by 20 ZCHF each as 20% of the Frankencoins are immediately redirected to the reserve after minting.

In case a minter cannot meet the repayment obligation and the subsequent liquidation results in a shortfall, it is in first priority covered by the minter reserve associated to the under-collateralized position, in second priority covered by equity, and in third priority covered by the collective minters reserve. While the balance sheet shows the Frankencoins minted against collateral assets, the provided collateral itself does not appear on the balance sheet as it belongs to the minter.

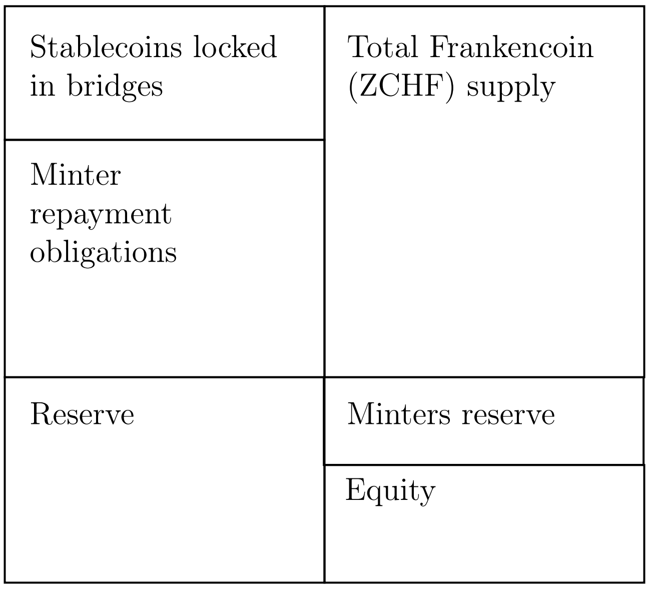

Stablecoins locked in bridges
Total Frankencoin (ZCHF) supply
Minter repayment obligations
Reserve
Minters reserve
Equity

Figure 2.A.1: Balance sheet. The balance sheet of the Frankencoin system.

2.B Basel Rules for Collateralized Lending

Haircuts are based on the type of exposure, see 22.4 in [CRE22]. For main equity indices and gold, the haircut is 15%, for other equities 25% (10 day holding period). Supervisors may also permit the banks to calculate their own haircuts.

Application of the comprehensive Approach

For repo-style transactions, supervisors may choose not to apply the haircuts specified in the comprehensive approach and may instead apply a haircut of zero, if the counterparty is a core market participant' as determined by the regulator. This would result in zero RWA because the collateralized mint plugin requires over-collateralization of all loans.

If the Frankencoin system is not exempt from the requirement that the counterparty is to be a core market participant, we may choose the parameters as follows.

$H_e = 0$	The exposure is in CHF (cash), hence no valuation risk
$H_c = 25\%/\sqrt{2}$	The 'other equity' haircut seems to be the most appropriate to reflect collateral in BTC. The 25% are for a 10-day holding period, hence we scale by $1/\sqrt{2}$ to have a 5-day haircut (which is the minimum allowed – our actual holding period is 1 day)

$H_{fx} = 0\%$	The asset is denominated in CHF, hence no additional currency risk

The above numbers are based on a 10-day holding period. If holding periods differ, the percentages are to be scaled by the square-root of time formula. However, for repo-style transactions, a minimum holding period of five days applies. Now, assuming a current loan to value ratio of $1/(1 + h^*)$, and an (artificial) holding period of 5 days, we use Equation (2.16) and set $E = 1$, $C = (1 + h^*)$:

$$E(h^*) = \max[0, 1 - (1 + h^*)(1 - H_c)], \tag{2.17}$$

$$= \max[0, H_c(1 + h^*) - h^*], \tag{2.18}$$

where the second line follows from algebra. For example, if $h^* > 0.21$, the resulting exposure is zero (using $Hc = 25\%/\sqrt{2}$, we set the second term in the max-term to zero and solve for h^*). Finally, the exposure E^* needs to be weighted by the counterparty risk weight. The counterparty in the Frankencoin system has generally no rating and cannot be forced to make up for losses in the system, hence the worst weight has to apply, which is w=150%, see Basel Committee on Banking Supervision (2017). Assuming that the loans are just at the (previously defined) liquidation level of $h = 10\%$, we arrive at RWA of

$$E(h)w = 0.09445 \cdot 1.5 = 14\%$$

of the loan amount, of which 8% are required as total capital. Hence, the capital requirement amounts to 1.1% of outstanding loans.

Comprehensive Approach: own Estimates for Haircuts

To calculate the haircuts, a 99th percentile, one-tailed confidence interval is to be used (see 22.50 of [CRE22]), and the choice of historical observation period (sample period) for calculating haircuts shall be a minimum of one year (see 22.53). The minimum holding period is to be set to 5 days. The VaR results in

$$VaR_q = (1 - \exp{(r_q)}) + k, \tag{2.19}$$

where q is the quantile (99% per Basel iii), r_q is the quantile return, and k the challenger fee. Using the historical data detailed in Appendix 2.C, the 5-day loss at the 99% quantile (using 1-day overlapping returns to calculate historical VaR at 1%-level) amounts to $19.44\%+k = 21.44\%$ (this compares to a haircut of $25\%/\sqrt{2} \approx 17.68\%$ used above). The resulting RWA, using Equation (2.18) and multiplying by the risk weight, at $h^* = 10\%$ is 20.38%, of which 8% are required as capital.[6] Hence, the capital requirement amounts to 1.6% of outstanding loans.

2.C Data

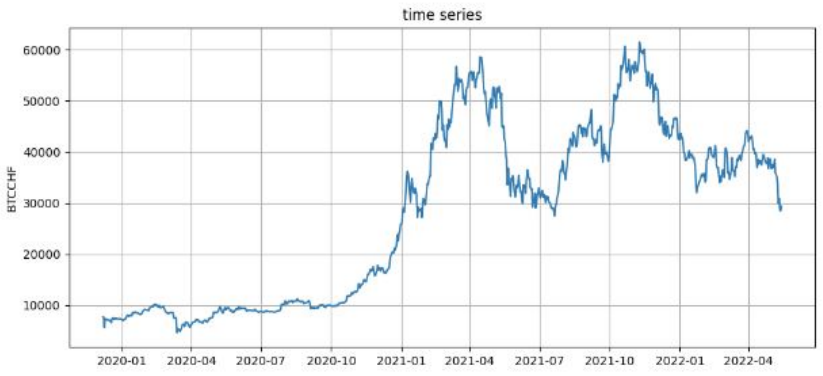


Figure 2.C.1: BTCCHF trade data. This figure plots the level data of the BTCCHF time-series. We have 890 observations of daily candle data without gaps from 2019-12-07 to 2022-05-14.

We gather 1-hour candle data from Kraken, consisting of a timestamp, open, low, high, close, number of trades, and volume.[7] From each open and close price we calculate 24h log-returns. Our return data has the following summary statistics. Table 2.9 tests the time-series of 24h log-returns for stationarity via Augmented Dickey-Fuller test and rejects the null-hypothesis of non-stationarity.

Figure 2.C.2 present a quantile-quantile plot against the normal distribution and a histogram. Figure 2.C.2 also analyses serial correlation of the 24h log-returns. The Autocorrelation Function shows a significant negative correlation at lag one, which however turns out to be driven from extreme returns, as we can see on plot (d).

[6]RWA $= E(h^*)w = (\mathrm{VaR}(1+h^*) - h^*)1.5$

[7]See support.kraken.com/hc/en-us/articles/360047124832-Downloadable-historical-OHLCVT-Open-High-Low-Close-Volume-Trades-data.

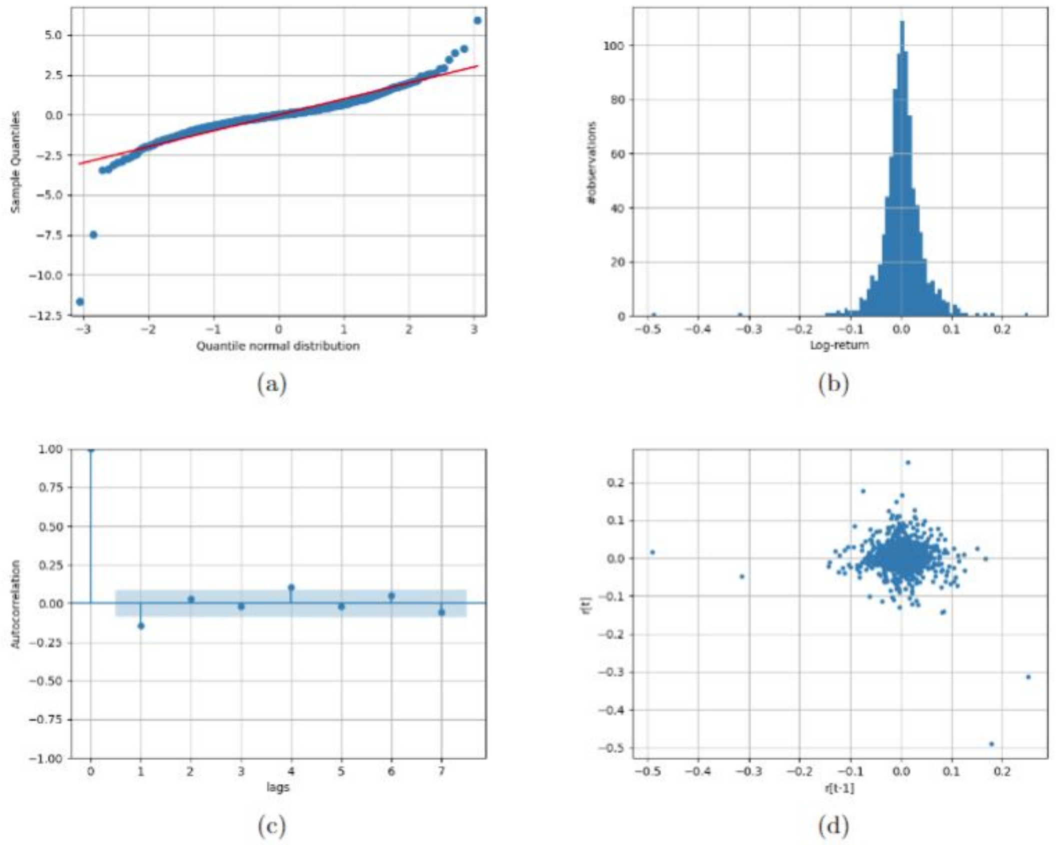


Figure 2.C.2: Return data from Kraken. Plot (a) shows a quantile-quantile plot of the 24h log-return data against a normal distribution, (b) plots a histogram of the 24h log-returns. From (a) we see that we have more extreme returns than what a normal distribution would suggest. In the center of the distribution, the returns move less than the normal distribution would imply. Plot (c) shows the autocorrelation function and a confidence band at the 0.95-level. Chart (d) plots the returns against the previous day return. The figure indicates that the significant autocorrelation at level 1 must be driven by rare extreme returns.

num. observations	890
min, max	[-49.09%, 25.13%]
mean	0.18%
variance	0.0018
skewness	-1.90
kurtosis	25.93

Table 2.C.1: Summary statistics for 24h log-return data.

ADF Statistic:	-13.8	
p-value:	0.000000	vspace15pt
Critical Value for 1%:	-3.4	

Table 2.C.2: Stationarity test. The ADF test suggests that the log-return data is stationary.

2.D Frankencoin (ZCHF) Token

The Frankencoin contract is an ERC-20 token with ticker ZCHF. It allows anyone to suggest new minter contracts that can mint Frankencoins under arbitrary rules. New minting contracts can be vetoed by qualified pool share holders. It also keeps the accounting for the minter's reserves when Frankencoins are minted or burned.

```
pragma solidity ^0.8.0;

import "./utils/ERC20PermitLight.sol";
import "./Equity.sol";
import "./interface/IReserve.sol";
import "./interface/IFrankencoin.sol";

/**
 * @title FrankenCoin
 * The Frankencoin (ZCHF) is an ERC-20 token that is designed to track the value of the Swiss franc.
 * It is not upgradable, but open to arbitrary minting plugins. These are automatically accepted if none of the
 * qualified pool share holders casts a veto, leading to a flexible but conservative governance.
 */
contract Frankencoin is ERC20PermitLight, IFrankencoin {
    /**
     * Minimal fee and application period when suggesting a new minter.
     */
    uint256 public constant MIN_FEE = 1000 * (10 ** 18);
    uint256 public immutable MIN_APPLICATION_PERIOD; // for example 10 days

    /**
     * The contract that holds the reserve.
     */
    IReserve public immutable override reserve;

    /**
     * How much of the reserve belongs to the minters. Everything else belongs to the pool share holders.
     * Stored with 6 additional digits of accuracy so no rounding is necessary when dealing with parts per
     * million (ppm) in reserve calculations.
     */
    uint256 private minterReserveE6;

    /**
```

```
     * Map of minters to approval time stamps. If the time stamp is in the past, the minter contract is allowed
     * to mint Frankencoins.
     */
    mapping(address minter => uint256 validityStart) public minters;

    /**
     * List of positions that are allowed to mint and the minter that registered them.
     */
    mapping(address position => address registeringMinter) public positions;

    event MinterApplied(address indexed minter, uint256 applicationPeriod, uint256 applicationFee, string message);
    event MinterDenied(address indexed minter, string message);
    event Loss(address indexed reportingMinter, uint256 amount, uint256 reserve);
    event Profit(address indexed minter, uint256 amount, uint256 reserve);

    error PeriodTooShort();
    error FeeTooLow();
    error AlreadyRegistered();
    error NotMinter();
    error TooLate();

    modifier minterOnly() {
        if (!isMinter(msg.sender) && !isMinter(positions[msg.sender])) revert NotMinter();
        _;
    }

    /**
     * Initiates the Frankencoin with the provided minimum application period for new plugins
     * in seconds, for example 10 days, i.e. 3600*24*10 = 864000
     */
    constructor(uint256 _minApplicationPeriod) ERC20(18) {
        MIN_APPLICATION_PERIOD = _minApplicationPeriod;
        reserve = new Equity(this);
    }

    function name() external pure override returns (string memory) {
        return "Frankencoin";
    }

    function symbol() external pure override returns (string memory) {
        return "ZCHF";
    }

    function initialize(address _minter, string calldata _message) external {
        require(totalSupply() == 0 && reserve.totalSupply() == 0);
        minters[_minter] = block.timestamp;
        emit MinterApplied(_minter, 0, 0, _message);
    }

    /**
     * Publicly accessible method to suggest a new way of minting Frankencoin.
     * @dev The caller has to pay an application fee that is irrevocably lost even if the new minter is vetoed.
     * The caller must assume that someone will veto the new minter unless there is broad consensus that the new minter
     * adds value to the Frankencoin system. Complex proposals should have application periods and applications fees
     * above the minimum. It is assumed that over time, informal ways to coordinate on new minters emerge. The message
     * parameter might be useful for initiating further communication. Maybe it contains a link to a website describing
     * the proposed minter.
     *
     * @param _minter              An address that is given the permission to mint Frankencoins
     * @param _applicationPeriod   The time others have to veto the suggestion, at least MIN_APPLICATION_PERIOD
     * @param _applicationFee      The fee paid by the caller, at least MIN_FEE
     * @param _message             An optional human readable message to everyone watching this contract
     */
    function suggestMinter(
        address _minter,
        uint256 _applicationPeriod,
        uint256 _applicationFee,
        string calldata _message
    ) external override {
        if (_applicationPeriod < MIN_APPLICATION_PERIOD) revert PeriodTooShort();
        if (_applicationFee < MIN_FEE) revert FeeTooLow();
        if (minters[_minter] != 0) revert AlreadyRegistered();
        _transfer(msg.sender, address(reserve), _applicationFee);
        minters[_minter] = block.timestamp + _applicationPeriod;
        emit MinterApplied(_minter, _applicationPeriod, _applicationFee, _message);
    }

    /**
     * Make the system more user friendly by skipping the allowance in many cases.
     * @dev We trust minters and the positions they have created to mint and burn as they please, so
     * giving them arbitrary allowances does not pose an additional risk.
     */
    function _allowance(address owner, address spender) internal view override returns (uint256) {
        uint256 explicit = super._allowance(owner, spender);
        if (explicit > 0) {
            return explicit; // don't waste gas checking minter
        } else if (isMinter(spender) || isMinter(getPositionParent(spender)) || spender == address(reserve)) {
            return INFINITY;
        } else {
            return 0;
        }
    }

    /**
     * The reserve provided by the owners of collateralized positions.
     * @dev The minter reserve can be used to cover losses after the equity holders have been wiped out.
     */
    function minterReserve() public view returns (uint256) {
        return minterReserveE6 / 1000000;
    }

    /**
     * Allows minters to register collateralized debt positions, thereby giving them the ability to mint Frankencoins.
     * @dev It is assumed that the responsible minter that registers the position ensures that the position can be trusted.
     */
    function registerPosition(address _position) external override {
        if (!isMinter(msg.sender)) revert NotMinter();
        positions[_position] = msg.sender;
    }

    /**
     * The amount of equity of the Frankencoin system in ZCHF, owned by the holders of Frankencoin Pool Shares.
     * @dev Note that the equity contract technically holds both the minter reserve as well as the equity, so the minter
     * reserve must be subtracted. All fees and other kind of income is added to the Equity contract and essentially
     * constitutes profits attributable to the pool share holders.
     */
    function equity() public view returns (uint256) {
        uint256 balance = balanceOf(address(reserve));
        uint256 minReserve = minterReserve();
        if (balance <= minReserve) {
            return 0;
        } else {
            return balance - minReserve;
        }
    }

    /**
     * Qualified pool share holders can deny minters during the application period.
     * @dev Calling this function is relatively cheap thanks to the deletion of a storage slot.
     */
    function denyMinter(address _minter, address[] calldata _helpers, string calldata _message) external override {
        if (block.timestamp > minters[_minter]) revert TooLate();
        reserve.checkQualified(msg.sender, _helpers);
        delete minters[_minter];
        emit MinterDenied(_minter, _message);
    }

    /**
     * Mints the provided amount of ZCHF to the target address, automatically forwarding
     * the minting fee and the reserve to the right place.
     */
    function mintWithReserve(
        address _target,
        uint256 _amount,
        uint32 _reservePPM,
        uint32 _feesPPM
    ) external override minterOnly {
        uint256 usableMint = (_amount * (1000_000 - _feesPPM - _reservePPM)) / 1000_000; // rounding down is fine
        _mint(_target, usableMint);
        _mint(address(reserve), _amount - usableMint); // rest goes to equity as reserves or as fees
        minterReserveE6 += _amount * _reservePPM;
    }

    function mint(address _target, uint256 _amount) external override minterOnly {
        _mint(_target, _amount);
    }

    /**
     * Anyone is allowed to burn their ZCHF.
     */
    function burn(uint256 _amount) external {
        _burn(msg.sender, _amount);
    }

    /**
     * Burn someone elses ZCHF.
     */
    function burnFrom(address _owner, uint256 _amount) external override minterOnly {
        _burn(_owner, _amount);
    }

    /**
     * Burn that amount without reclaiming the reserve, but freeing it up and thereby essentially donating it to the
     * pool share holders. This can make sense in combination with 'notifyLoss', i.e. when it is the pool share
     * holders that bear the risk and depending on the outcome they make a profit or a loss.
     *
     * Design rule: Minters calling this method are only allowed to so for tokens amounts they previously minted with
     * the same _reservePPM amount.
     *
     * For example, if someone minted 50 ZCHF earlier with a 20% reserve requirement (200000 ppm), they got 40 ZCHF
     * and paid 10 ZCHF into the reserve. Now they want to repay the debt by burning 50 ZCHF. When doing so using this
     * method, 50 ZCHF get burned and on top of that, 10 ZCHF previously assigned to the minter's reserved are
     * reassigned to the pool share holders.
     */
    function burnWithoutReserve(uint256 amount, uint32 reservePPM) public override minterOnly {
        _burn(msg.sender, amount);
        uint256 reserveReduction = amount * reservePPM;
        if (reserveReduction > minterReserveE6) {
            emit Profit(msg.sender, minterReserveE6, 0);
            minterReserveE6 = 0; // should never happen, but we want robust behavior in case it does
        } else {
            minterReserveE6 -= reserveReduction;
            emit Profit(msg.sender, reserveReduction, minterReserveE6);
        }
    }

    /**
     * Burns the provided number of tokens plus whatever reserves are associated with that amount given the reserve
     * requirement. The caller is only allowed to use this method for tokens also minted through the caller with the
     * same _reservePPM amount.
     *
     * Example: the calling contract has previously minted 100 ZCHF with a reserve ratio of 20% (i.e. 200000 ppm).
     * Now they have 41 ZCHF that they do not need so they decide to repay that amount. Assuming the reserves are
     * only 90% covered, the call to burnWithReserve will burn the 41 plus 9 from the reserve, reducing the outstanding
     * 'debt' of the caller by 50 ZCHF in total. This total is returned by the method so the caller knows how much less
     * they owe.
     */
    function burnWithReserve(
        uint256 _amountExcludingReserve,
        uint32 _reservePPM
    ) external override minterOnly returns (uint256) {
        uint256 freedAmount = calculateFreedAmount(_amountExcludingReserve, _reservePPM); // 50 in the example
        minterReserveE6 -= freedAmount * _reservePPM; // reduce reserve requirements by original ratio
        _transfer(address(reserve), msg.sender, freedAmount - _amountExcludingReserve); // collect assigned reserve
        _burn(msg.sender, freedAmount); // burn the rest of the freed amount
        return freedAmount;
    }

    /**
     * Burns the target amount taking the tokens to be burned from the payer and the payer's reserve.
     * Only use this method for tokens also minted by the caller with the same _reservePPM.
     *
     * Example: the calling contract has previously minted 100 ZCHF with a reserve ratio of 20% (i.e. 200000 ppm).
     * To burn half of that again, the minter calls burnFrom with a target amount of 50 ZCHF. Assuming that reserves
     * are only 90% covered, this call will deduct 41 ZCHF from the payer's balance and 9 from the reserve, while
     * reducing the minter reserve by 10.
     */
    function burnFromWithReserve(
        address payer,
        uint256 targetTotalBurnAmount,
        uint32 reservePPM
    ) external override minterOnly returns (uint256) {
        uint256 assigned = calculateAssignedReserve(targetTotalBurnAmount, reservePPM);
        _transfer(address(reserve), payer, assigned); // send reserve to owner
        _burn(payer, targetTotalBurnAmount); // and burn the full amount from the owner's address
        minterReserveE6 -= targetTotalBurnAmount * reservePPM; // reduce reserve requirements by original ratio
        return assigned;
    }

    /**
     * Calculates the reserve attributable to someone who minted the given amount with the given reserve requirement.
     * Under normal circumstances, this is just the reserver requirement multiplied by the amount. However, after a
     * severe loss of capital that burned into the minter's reserve, this can also be less than that.
     */
    function calculateAssignedReserve(uint256 mintedAmount, uint32 _reservePPM) public view returns (uint256) {
        uint256 theoreticalReserve = (_reservePPM * mintedAmount) / 1000000;
        uint256 currentReserve = balanceOf(address(reserve));
        uint256 minterReserve_ = minterReserve();
        if (currentReserve < minterReserve_) {
            // not enough reserves, owner has to take a loss
            return (theoreticalReserve * currentReserve) / minterReserve_;
        } else {
            return theoreticalReserve;
        }
    }

    /**
     * Calculate the amount that is freed when returning amountExcludingReserve given a reserve ratio of reservePPM,
     * taking into account potential losses. Example values in the comments.
     */
    function calculateFreedAmount(
        uint256 amountExcludingReserve /* 41 */,
        uint32 reservePPM /* 20% */
    ) public view returns (uint256) {
        uint256 currentReserve = balanceOf(address(reserve)); // 18, 10% below what we should have
        uint256 minterReserve_ = minterReserve(); // 20
        uint256 adjustedReservePPM = currentReserve < minterReserve_
            ? (reservePPM * currentReserve) / minterReserve_
            : reservePPM; // 18%
        return (1000000 * amountExcludingReserve) / (1000000 - adjustedReservePPM); // 41 / (1-18%) = 50
    }

    /**
     * Notify the Frankencoin that a minter lost economic access to some coins. This does not mean that the coins are
     * literally lost. It just means that some ZCHF will likely never be repaid and that in order to bring the system
     * back into balance, the lost amount of ZCHF must be removed from the reserve instead.
     *
     * For example, if a minter printed 1 million ZCHF for a mortgage and the mortgage turned out to be unsound with
     * the house only yielding 800'000 in the subsequent auction, there is a loss of 200'000 that needs to be covered
     * by the reserve.
     */
    function notifyLoss(uint256 _amount) external override minterOnly {
        uint256 reserveLeft = balanceOf(address(reserve));
        if (reserveLeft >= _amount) {
            _transfer(address(reserve), msg.sender, _amount);
        } else {
            _transfer(address(reserve), msg.sender, reserveLeft);
            _mint(msg.sender, _amount - reserveLeft);
        }
        emit Loss(msg.sender, _amount, reserveLeft);
    }

    /**
     * Returns true if the address is an approved minter.
     */
    function isMinter(address _minter) public view override returns (bool) {
        return minters[_minter] != 0 && block.timestamp >= minters[_minter];
    }

    /**
```

```
     * Returns the address of the minter that created this position or null if the provided address is unknown.
     */
    function getPositionParent(address _position) public view override returns (address) {
        return positions[_position];
    }
}
```

2.E Frankencoin Pool Share (FPS) Token

The Equity contract is an ERC-20 token with name *Frankencoin Pool Shares* ticker FPS. It contains an automated market maker that governs the issuance and redemption of pool shares, keeps track of how long pool shares have been held on the same address, and contains the governance logic.

```
pragma solidity ^0.8.0;

import "./Frankencoin.sol";
import "./utils/MathUtil.sol";
import "./interface/IReserve.sol";
import "./interface/IERC677Receiver.sol";

/**
 * @title Equity
 * If the Frankencoin system was a bank, this contract would represent the equity on its balance sheet.
 * Like with a corporation, the owners of the equity capital are the shareholders, or in this case the holders
 * of Frankencoin Pool Shares (FPS) tokens. Anyone can mint additional FPS tokens by adding Frankencoins to the
 * reserve pool. Also, FPS tokens can be redeemed for Frankencoins again after a minimum holding period.
 * Furthermore, the FPS shares come with some voting power. Anyone that held at least 3% of the holding-period-
 * weighted reserve pool shares gains veto power and can veto new proposals.
 */
contract Equity is ERC20PermitLight, MathUtil, IReserve {
    /**
     * The VALUATION_FACTOR determines the market cap of the reserve pool shares relative to the equity reserves.
     * The following always holds: Market Cap = Valuation Factor * Equity Reserve = Price * Supply
     *
     * In the absence of profits and losses, the variables grow as follows when FPS tokens are minted:
     *
     * |   Reserve     |   Market Cap  |     Price     |     Supply   |
     * |          1000 |          3000 |             3 |         1000 |
     * |       1000000 |       3000000 |           300 |        10000 |
     * |    1000000000 |    3000000000 |         30000 |       100000 |
     * | 1000000000000 | 3000000000000 |       3000000 |      1000000 |
     *
     * I.e., the supply is proporational to the cubic root of the reserve and the price is proportional to the
     * squared cubic root. When profits accumulate or losses materialize, the reserve, the market cap,
     * and the price are adjusted proportionally, with the supply staying constant. In the absence of an extreme
     * inflation of the Swiss franc, it is unlikely that there will ever be more than ten million FPS.
     */
    uint32 public constant VALUATION_FACTOR = 3;

    uint256 private constant MINIMUM_EQUITY = 1000 * ONE_DEC18;

    /**
     * The quorum in basis points. 100 is 1%.
     */
    uint32 private constant QUORUM = 200;

    /**
     * The number of digits to store the average holding time of share tokens.
     */
    uint8 private constant TIME_RESOLUTION_BITS = 20;

    /**
     * The minimum holding duration. You are not allowed to redeem your pool shares if you held them
     * for less than the minimum holding duration at average. For example, if you have two pool shares on your
     * address, one acquired 5 days ago and one acquired 105 days ago, you cannot redeem them as the average
     * holding duration of your shares is only 55 days < 90 days.
     */
    uint256 public constant MIN_HOLDING_DURATION = 90 days << TIME_RESOLUTION_BITS; // Set to 5 for local testing

    Frankencoin public immutable zchf;

    /**
     * @dev To track the total number of votes we need to know the number of votes at the anchor time and when the
     * anchor time was. This is (hopefully) stored in one 256 bit slot, with the anchor time taking 64 Bits and
     * the total vote count 192 Bits. Given the sub-second resolution of 20 Bits, the implicit assumption is
     * that the timestamp can always be stored in 44 Bits (i.e. it does not exceed half a million years). Further,
     * given 18 decimals (about 60 Bits), this implies that the total supply cannot exceed
     *   192 - 60 - 44 - 20 = 68 Bits
     * Here, we are also save, as 68 Bits would imply more than a trillion outstanding shares. In fact,
     * a limit of about 2**36 shares (that's about 2**96 Bits when taking into account the decimals) is imposed
     * when minting. This means that the maximum supply is billions shares, which is could only be reached in
     * a scenario with hyper inflation, in which case the stablecoin is worthless anyway.
     */
    uint192 private totalVotesAtAnchor; // Total number of votes at the anchor time, see comment on the um
    uint64 private totalVotesAnchorTime; // 44 Bit for the time stamp, 20 Bit sub-second time resolution

    /**
     * Keeping track on who delegated votes to whom.
     * Note that delegation does not mean you cannot vote / veto any more, it just means that the delegate can
     * benefit from your votes when invoking a veto. Circular delegations are valid, do not help when voting.
     */
    mapping(address owner => address delegate) public delegates;

    /**
     * A time stamp in the past such that: votes = balance * (time passed since anchor was set)
     */
    mapping(address owner => uint64 timestamp) private voteAnchor; // 44 bits for time stamp, 20 subsecond resolution

    event Delegation(address indexed from, address indexed to); // indicates a delegation
    event Trade(address who, int amount, uint totPrice, uint newprice); // amount pos or neg for mint or redemption

    constructor(Frankencoin zchf_) ERC20(18) {
        zchf = zchf_;
    }

    function name() external pure override returns (string memory) {
        return "Frankencoin Pool Share";
    }

    function symbol() external pure override returns (string memory) {
        return "FPS";
    }

    /**
     * Returns the price of one FPS in ZCHF with 18 decimals precision.
     */
    function price() public view returns (uint256) {
        uint256 equity = zchf.equity();
        if (equity == 0 || totalSupply() == 0) {
            return ONE_DEC18; // initial price is 1000 ZCHF for the first 1000 FPS
        } else {
            return (VALUATION_FACTOR * zchf.equity() * ONE_DEC18) / totalSupply();
        }
    }

    function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
        super._beforeTokenTransfer(from, to, amount);
        if (amount > 0) {
            // No need to adjust the sender votes. When they send out 10% of their shares, they also lose 10% of
            // their votes so everything falls nicely into place. Recipient votes should stay the same, but grow
            // faster in the future, requiring an adjustment of the anchor.
            uint256 roundingLoss = _adjustRecipientVoteAnchor(to, amount);
            // The total also must be adjusted and kept accurate by taking into account the rounding error.
            _adjustTotalVotes(from, amount, roundingLoss);
        }
    }

    /**
```

```
     * Returns whether the given address is allowed to redeem FPS, which is the
     * case after their average holding duration is larger than the required minimum.
     */
    function canRedeem(address owner) public view returns (bool) {
        return _anchorTime() - voteAnchor[owner] >= MIN_HOLDING_DURATION;
    }

    /**
     * Decrease the total votes anchor when tokens lose their voting power due to being moved
     * @param from      sender
     * @param amount    amount to be sent
     */
    function _adjustTotalVotes(address from, uint256 amount, uint256 roundingLoss) internal {
        uint64 time = _anchorTime();
        uint256 lostVotes = from == address(0x0) ? 0 : (time - voteAnchor[from]) * amount;
        totalVotesAtAnchor = uint192(totalVotes() - roundingLoss - lostVotes);
        totalVotesAnchorTime = time;
    }

    /**
     * the vote anchor of the recipient is moved forward such that the number of calculated
     * votes does not change despite the higher balance.
     * @param to        receiver address
     * @param amount    amount to be received
     * @return the number of votes lost due to rounding errors
     */
    function _adjustRecipientVoteAnchor(address to, uint256 amount) internal returns (uint256) {
        if (to != address(0x0)) {
            uint256 recipientVotes = votes(to); // for example 21 if 7 shares were held for 3 seconds
            uint256 newbalance = balanceOf(to) + amount; // for example 11 if 4 shares are added
            // new example anchor is only 21 / 11 = 1 second in the past
            voteAnchor[to] = uint64(_anchorTime() - recipientVotes / newbalance);
            return recipientVotes % newbalance; // we have lost 21 % 11 = 10 votes
        } else {
            // optimization for burn, vote anchor of null address does not matter
            return 0;
        }
    }

    /**
     * Time stamp with some additional bits for higher resolution.
     */
    function _anchorTime() internal view returns (uint64) {
        return uint64(block.timestamp << TIME_RESOLUTION_BITS);
    }

    /**
     * The relative voting power of the address.
     * @return A percentage with 1e18 being 100%
     */
    function relativeVotes(address holder) external view returns (uint256) {
        return (ONE_DEC18 * votes(holder)) / totalVotes();
    }

    /**
     * The votes of the holder, excluding votes from delegates.
     */
    function votes(address holder) public view returns (uint256) {
        return balanceOf(holder) * (_anchorTime() - voteAnchor[holder]);
    }

    /**
     * Total number of votes in the system.
     */
    function totalVotes() public view returns (uint256) {
        return totalVotesAtAnchor + totalSupply() * (_anchorTime() - totalVotesAnchorTime);
    }

    /**
     * The number of votes the sender commands when taking the support of the helpers into account.
     * @param sender    The address whose total voting power is of interest
     * @param helpers   An incrementally sorted list of helpers without duplicates and without the sender.
     *                  The call fails if the list contains an address that does not delegate to sender.
     *                  For indirect delegates, i.e. a -> b -> c, both a and b must be included for both to count.
     * @return          The total number of votes of sender at the current point in time.
     */
    function votesDelegated(address sender, address[] calldata helpers) public view returns (uint256) {
        uint256 _votes = votes(sender);
        require(_checkDuplicatesAndSorted(helpers));
        for (uint i = 0; i < helpers.length; i++) {
            address current = helpers[i];
            require(current != sender);
            require(_canVoteFor(sender, current));
            _votes += votes(current);
        }
        return _votes;
    }

    function _checkDuplicatesAndSorted(address[] calldata helpers) internal pure returns (bool ok) {
        if (helpers.length <= 1) {
            return true;
        } else {
            address prevAddress = helpers[0];
            for (uint i = 1; i < helpers.length; i++) {
                if (helpers[i] <= prevAddress) {
                    return false;
                }
                prevAddress = helpers[i];
            }
            return true;
        }
    }

    /**
     * Checks whether the sender address is qualified given a list of helpers that delegated their votes
     * directly or indirectly to the sender. It is the responsiblity of the caller to figure out whether
     * helpes are necessary and to identify them by scanning the blockchain for Delegation events.
     */
    function checkQualified(address sender, address[] calldata helpers) public view override {
        uint256 _votes = votesDelegated(sender, helpers);
        if (_votes * 10000 < QUORUM * totalVotes()) revert NotQualified();
    }

    error NotQualified();

    /**
     * Increases the voting power of the delegate by your number of votes without taking away any voting power
     * from the sender.
     */
    function delegateVoteTo(address delegate) external {
        delegates[msg.sender] = delegate;
        emit Delegation(msg.sender, delegate);
    }

    function _canVoteFor(address delegate, address owner) internal view returns (bool) {
        if (owner == delegate) {
            return true;
        } else if (owner == address(0x0)) {
            return false;
        } else {
            return _canVoteFor(delegate, delegates[owner]);
        }
    }

    /**
     * Since quorum is rather low, it is important to have a way to prevent malicious minority holders
     * from blocking the whole system. This method provides a way for the good guys to team up and destroy
     * the bad guy's votes (at the cost of also reducing their own votes). This mechanism potentially
     * gives full control over the system to whoever has 51% of the votes.
     *
     * Since this is a rather aggressive measure, delegation is not supported. Every holder must call this
     * method on their own.
     * @param targets   The target addresses to remove votes from
     * @param votesToDestroy    The maximum number of votes the caller is willing to sacrifice
     */
    function kamikaze(address[] calldata targets, uint256 votesToDestroy) external {
        uint256 budget = _reduceVotes(msg.sender, votesToDestroy);
        uint256 destroyedVotes = 0;
        for (uint256 i = 0; i < targets.length && destroyedVotes < budget; i++) {
            destroyedVotes += _reduceVotes(targets[i], budget - destroyedVotes);
        }
        require(destroyedVotes > 0); // sanity check
        totalVotesAtAnchor = uint192(totalVotes() - destroyedVotes - budget);
        totalVotesAnchorTime = _anchorTime();
    }

    function _reduceVotes(address target, uint256 amount) internal returns (uint256) {
        uint256 votesBefore = votes(target);
        if (amount >= votesBefore) {
            amount = votesBefore;
            voteAnchor[target] = _anchorTime();
            return votesBefore;
        } else {
            voteAnchor[target] = uint64(_anchorTime() - (votesBefore - amount) / balanceOf(target));
            return votesBefore - votes(target);
        }
    }

    /**
     * Call this method to obtain newly minted pool shares in exchange for Frankencoins.
     * No allowance required (i.e. it is hardcoded in the Frankencoin token contract).
     * Make sure to invest at least 10e-12 * market cap to avoid rounding losses.
     *
     * @dev If equity is close to zero or negative, you need to send enough ZCHF to bring equity back to 1000 ZCHF.
     *
     * @param amount            Frankencoins to invest
     * @param expectedShares    Minimum amount of expected shares for frontrunning protection
     */
    function invest(uint256 amount, uint256 expectedShares) external returns (uint256) {
        zchf.transferFrom(msg.sender, address(this), amount);
        uint256 equity = zchf.equity();
        require(equity >= MINIMUM_EQUITY, "insuf equity"); // ensures that the initial deposit is at least 1000 ZCHF

        uint256 shares = _calculateShares(equity <= amount ? 0 : equity - amount, amount);
        require(shares >= expectedShares);
        _mint(msg.sender, shares);
        emit Trade(msg.sender, int(shares), amount, price());

        // limit the total supply to a reasonable amount to guard against overflows with price and vote calculations
        // the 36 bits are 68 bits for magnitude and 60 bits for precision, as calculated in an above comment
        require(totalSupply() <= type(uint96).max, "total supply exceeded");
        return shares;
    }

    /**
     * Calculate shares received when investing Frankencoins
     * @param investment    ZCHF to be invested
     * @return shares to be received in return
     */
    function calculateShares(uint256 investment) external view returns (uint256) {
        return _calculateShares(zchf.equity(), investment);
    }

    function _calculateShares(uint256 capitalBefore, uint256 investment) internal view returns (uint256) {
        uint256 totalShares = totalSupply();
        uint256 investmentExFees = (investment * 997) / 1000; // remove 0.3% fee
        // Assign 1000 FPS for the initial deposit, calculate the amount otherwise
        uint256 newTotalShares = capitalBefore < MINIMUM_EQUITY || totalShares == 0
            ? totalShares + 1000 * ONE_DEC18
            : _mulD18(totalShares, _cubicRoot(_divD18(capitalBefore + investmentExFees, capitalBefore)));
        return newTotalShares - totalShares;
    }

    /**
     * Redeem the given amount of shares owned by the sender and transfer the proceeds to the target.
     * @return The amount of ZCHF transferred to the target
     */
    function redeem(address target, uint256 shares) external returns (uint256) {
        return _redeemFrom(msg.sender, target, shares);
    }

    /**
     * Like redeem(...), but with an extra parameter to protect against frontrunning.
     * @param expectedProceeds  The minimum acceptable redemption proceeds.
     */
    function redeemExpected(address target, uint256 shares, uint256 expectedProceeds) external returns (uint256) {
        uint256 proceeds = _redeemFrom(msg.sender, target, shares);
        require(proceeds >= expectedProceeds);
        return proceeds;
    }

    /**
     * Redeem FPS based on an allowance from the owner to the caller.
     * See also redeemExpected(...).
     */
    function redeemFrom(
        address owner,
        address target,
        uint256 shares,
        uint256 expectedProceeds
    ) external returns (uint256) {
        _useAllowance(owner, msg.sender, shares);
        uint256 proceeds = _redeemFrom(owner, target, shares);
        require(proceeds >= expectedProceeds);
        return proceeds;
    }

    function _redeemFrom(address owner, address target, uint256 shares) internal returns (uint256) {
        require(canRedeem(owner));
        uint256 proceeds = calculateProceeds(shares);
        _burn(owner, shares);
        zchf.transfer(target, proceeds);
        emit Trade(owner, -int(shares), proceeds, price());
        return proceeds;
    }

    /**
     * Calculate ZCHF received when depositing shares
     * @param shares number of shares we want to exchange for ZCHF,
     *               in dec18 format
     * @return amount of ZCHF received for the shares
     */
    function calculateProceeds(uint256 shares) public view returns (uint256) {
        uint256 totalShares = totalSupply();
        require(shares + ONE_DEC18 < totalShares, "too many shares"); // make sure there is always at least one share
        uint256 capital = zchf.equity();
        uint256 reductionAfterFees = (shares * 997) / 1000;
        uint256 newCapital = _mulD18(capital, _power3(_divD18(totalShares - reductionAfterFees, totalShares)));
        return capital - newCapital;
    }

    /**
     * If there is less than 1000 ZCHF in equity left (maybe even negative), the system is at risk
     * and we should allow qualified FPS holders to restructure the system.
     *
     * Example: there was a devastating loss and equity stands at -1'000'000. Most shareholders have lost hope in the
     * Frankencoin system except for a group of small FPS holders who still believes in it and is willing to provide
     * 2'000'000 ZCHF to save it. These brave souls are essentially donating 1'000'000 to the minter reserve and it
     * would be wrong to force them to share the other million with the passive FPS holders. Instead, they will get
     * the possibility to bootstrap the system again owning 100% of all FPS shares.
     *
     * @param helpers          A list of addresses that delegate to the caller in incremental order
     * @param addressesToWipe  A list of addresses whose FPS will be burned to zero
     */
    function restructureCapTable(address[] calldata helpers, address[] calldata addressesToWipe) external {
        require(zchf.equity() < MINIMUM_EQUITY);
        checkQualified(msg.sender, helpers);
        for (uint256 i = 0; i < addressesToWipe.length; i++) {
            address current = addressesToWipe[i];
            _burn(current, balanceOf(current));
        }
    }
}
```

2.F Minting Hub

The minting hub is a minter contract that serves as a basis to open collateralized positions with the Frankencoin's unique collateralized, oracle-free minting mechanism. The minting hub allows users to open new positions, to clone existing positions, to launch challenges, and to bid on challenges.

```
pragma solidity ^0.8.0;

import "./interface/IERC20.sol";
import "./interface/IReserve.sol";
import "./interface/IFrankencoin.sol";
import "./interface/IPosition.sol";
import "./interface/IPositionFactory.sol";

/**
 * @title Minting Hub
 * The central hub for creating, cloning and challenging collateralized Frankencoin positions.
 * @dev Only one instance of this contract is required, whereas every new position comes with a new position
 * contract. Pending challenges are stored as structs in an array.
 */
contract MintingHub {
    /**
     * Irrevocable fee in ZCHF when proposing a new position (but not when cloning an existing one).
     */
    uint256 public constant OPENING_FEE = 1000 * 10 ** 18;

    /**
     * The challenger reward in parts per million (ppm) relative to the challenged amount, whereas
     * challenged amount if defined as the challenged collateral amount times the liquidation price.
     */
    uint32 public constant CHALLENGER_REWARD = 20000; // 2%

    IPositionFactory private immutable POSITION_FACTORY; // position contract to clone

    IFrankencoin public immutable zchf; // currency
    Challenge[] public challenges; // list of open challenges

    /**
     * Map to remember pending postponed collateral returns.
     * @dev It maps collateral => beneficiary => amount.
     */
    mapping(address collateral => mapping(address owner => uint256 amount)) public pendingReturns;

    struct Challenge {
        address challenger; // the address from which the challenge was initiated
        uint64 start; // the start of the challenge
        IPosition position; // the position that was challenged
        uint256 size; // how much collateral the challenger provided
    }

    event PositionOpened(
        address indexed owner,
        address indexed position,
        address zchf,
        address collateral,
        uint256 price
    );
    event ChallengeStarted(address indexed challenger, address indexed position, uint256 size, uint256 number);
    event ChallengeAverted(address indexed position, uint256 number, uint256 size);
    event ChallengeSucceeded(
        address indexed position,
        uint256 number,
        uint256 bid,
        uint256 acquiredCollateral,
        uint256 challengeSize
    );
    event PostPonedReturn(address collateral, address indexed beneficiary, uint256 amount);

    error UnexpectedPrice();
    error InvalidPos();

    modifier validPos(address position) {
        if (zchf.getPositionParent(position) != address(this)) revert InvalidPos();
        _;
    }

    constructor(address _zchf, address _factory) {
        zchf = IFrankencoin(_zchf);
        POSITION_FACTORY = IPositionFactory(_factory);
    }

    function openPositionOneWeek(
        address _collateralAddress,
        uint256 _minCollateral,
        uint256 _initialCollateral,
        uint256 _mintingMaximum,
        uint256 _expirationSeconds,
        uint64 _challengeSeconds,
        uint32 _annualInterestPPM,
        uint256 _liqPrice,
        uint32 _reservePPM
    ) public returns (address) {
        return
            openPosition(
                _collateralAddress,
                _minCollateral,
                _initialCollateral,
                _mintingMaximum,
                7 days,
                _expirationSeconds,
                _challengeSeconds,
                _annualInterestPPM,
                _liqPrice,
                _reservePPM
            );
    }

    /**
     * Open a collateralized loan position. See also https://docs.frankencoin.com/positions/open .
     * @dev For a successful call, you must set an allowance for the collateral token, allowing
     * the minting hub to transfer the initial collateral amount to the newly created position and to
     * withdraw the fees.
     *
     * @param _collateralAddress        address of collateral token
     * @param _minCollateral     minimum collateral required to prevent dust amounts
     * @param _initialCollateral amount of initial collateral to be deposited
     * @param _mintingMaximum    maximal amount of ZCHF that can be minted by the position owner
     * @param _expirationSeconds position tenor in unit of timestamp (seconds) from 'now'
     * @param _challengeSeconds  challenge period. Longer for less liquid collateral.
     * @param _annualInterestPPM ppm of minted amount that is paid as fee for each year of duration
     * @param _liqPrice          Liquidation price with (36 - token decimals) decimals,
     *                           e.g. 18 decimals for an 18 dec collateral, 36 decs for a 0 dec collateral.
     * @param _reservePPM        ppm of minted amount that is locked as borrower's reserve, e.g. 20%
     * @return address           address of created position
     */
    function openPosition(
        address _collateralAddress,
        uint256 _minCollateral,
        uint256 _initialCollateral,
        uint256 _mintingMaximum,
        uint256 _initPeriodSeconds,
        uint256 _expirationSeconds,
        uint64 _challengeSeconds,
        uint32 _annualInterestPPM,
        uint256 _liqPrice,
        uint32 _reservePPM
    ) public returns (address) {
        require(_annualInterestPPM <= 1000000);
        require(_reservePPM <= 1000000);
        require(IERC20(_collateralAddress).decimals() <= 24); // leaves 12 digits for price
        require(_initialCollateral >= _minCollateral, "must start with min col");
        require(_minCollateral * _liqPrice >= 5000 ether * 10 ** 18); // must start with at least 5000 ZCHF worth of collateral
        IPosition pos = IPosition(
            POSITION_FACTORY.createNewPosition(
                msg.sender,
                address(zchf),
                _collateralAddress,
                _minCollateral,
                _mintingMaximum,
                _initPeriodSeconds,
                _expirationSeconds,
                _challengeSeconds,
                _annualInterestPPM,
                _liqPrice,
                _reservePPM
            )
        );
        zchf.registerPosition(address(pos));
        zchf.transferFrom(msg.sender, address(zchf.reserve()), OPENING_FEE);
        IERC20(_collateralAddress).transferFrom(msg.sender, address(pos), _initialCollateral);

        emit PositionOpened(msg.sender, address(pos), address(zchf), _collateralAddress, _liqPrice);
        return address(pos);
    }

    /**
     * Clones an existing position and immediately tries to mint the specified amount using the given collateral.
     * @dev This needs an allowance to be set on the collateral contract such that the minting hub can get the collateral.
     */
    function clonePosition(
        address position,
        uint256 _initialCollateral,
        uint256 _initialMint,
        uint256 expiration
    ) public validPos(position) returns (address) {
        IPosition existing = IPosition(position);
        require(expiration <= IPosition(existing.original()).expiration());
        existing.reduceLimitForClone(_initialMint);
        address pos = POSITION_FACTORY.clonePosition(position);
        zchf.registerPosition(pos);
        IPosition(pos).initializeClone(msg.sender, existing.price(), _initialCollateral, _initialMint, expiration);
        existing.collateral().transferFrom(msg.sender, pos, _initialCollateral);

        emit PositionOpened(
            msg.sender,
            address(pos),
            address(zchf),
            address(IPosition(pos).collateral()),
            IPosition(pos).price()
        );
        return address(pos);
    }

    /**
     * Launch a challenge (Dutch auction) on a position
     * @param _positionAddr      address of the position we want to challenge
     * @param _collateralAmount  size of the collateral we want to challenge (dec 18)
     * @param expectedPrice      position.price() to guard against the minter fruntrunning with a price change
     * @return index of the challenge in challenge-array
     */
    function launchChallenge(
        address _positionAddr,
        uint256 _collateralAmount,
        uint256 expectedPrice
    ) external validPos(_positionAddr) returns (uint256) {
        IPosition position = IPosition(_positionAddr);
        if (position.price() != expectedPrice) revert UnexpectedPrice();
        IERC20(position.collateral()).transferFrom(msg.sender, address(this), _collateralAmount);
        uint256 pos = challenges.length;
        challenges.push(Challenge(msg.sender, uint64(block.timestamp), position, _collateralAmount));
        position.notifyChallengeStarted(_collateralAmount);
        emit ChallengeStarted(msg.sender, address(position), _collateralAmount, pos);
        return pos;
    }

    /**
     * Post a bid in ZCHF given an open challenge.
     *
     * @dev In case that the collateral cannot be transfered back to the challenger (i.e. because the collateral token
     * has a blacklist and the challenger is on it), it is possible to postpone the return of the collateral.
     *
     * @param _challengeNumber  index of the challenge as broadcast in the event
     * @param size              how much of the collateral the caller wants to bid for at most
     *                          (automatically reduced to the available amount)
     * @param postponeCollateralReturn To postpone the return of the collateral to the challenger. Usually false.
     */
    function bid(uint32 _challengeNumber, uint256 size, bool postponeCollateralReturn) external {
        Challenge memory challenge = challenges[_challengeNumber];
        (uint256 liqPrice, uint64 phase1, uint64 phase2) = challenge.position.challengeData();
        size = challenge.size < size ? challenge.size : size; // cannot bid for more than the size of the challenge

        if (block.timestamp <= challenge.start + phase1) {
            _avertChallenge(challenge, _challengeNumber, liqPrice, size);
            emit ChallengeAverted(address(challenge.position), _challengeNumber, size);
        } else {
            _returnChallengerCollateral(challenge, _challengeNumber, size, postponeCollateralReturn);
            (uint256 transferredCollateral, uint256 offer) = _finishChallenge(
                challenge,
                liqPrice,
                phase1,
                phase2,
                size
            );
            emit ChallengeSucceeded(address(challenge.position), _challengeNumber, offer, transferredCollateral, size);
        }
    }

    function _finishChallenge(
        Challenge memory challenge,
        uint256 liqPrice,
        uint64 phase1,
        uint64 phase2,
        uint256 size
    ) internal returns (uint256, uint256) {
        // Repayments depend on what was actually minted, whereas bids depend on the available collateral
        (address owner, uint256 collateral, uint256 repayment, uint32 reservePPM) = challenge
            .position
            .notifyChallengeSucceeded(msg.sender, size);

        // No overflow possible thanks to invariant (col * price <= limit * 10**18)
        // enforced in Position.setPrice and knowing that collateral <= col.
        uint256 offer = (_calculatePrice(challenge.start + phase1, phase2, liqPrice) * collateral) / 10 ** 18;
        zchf.transferFrom(msg.sender, address(this), offer); // get money from bidder
        uint256 reward = (offer * CHALLENGER_REWARD) / 1000_000;
        uint256 fundsNeeded = reward + repayment;

        if (offer > fundsNeeded) {
            zchf.transfer(owner, offer - fundsNeeded);
        } else if (offer < fundsNeeded) {
            zchf.notifyLoss(fundsNeeded - offer); // ensure we have enough to pay everything
        }
        zchf.transfer(challenge.challenger, reward); // pay out the challenger reward
        zchf.burnWithoutReserve(repayment, reservePPM); // Repay the challenged part
        return (collateral, offer);
    }

    function _avertChallenge(Challenge memory challenge, uint32 number, uint256 liqPrice, uint256 size) internal {
        if (msg.sender == challenge.challenger) {
            // allow challenger to cancel challenge without paying themselves
        } else {
            zchf.transferFrom(msg.sender, challenge.challenger, (size * liqPrice) / (10 ** 18));
        }

        challenge.position.notifyChallengeAverted(size);
        challenge.position.collateral().transfer(msg.sender, size);
        if (size < challenge.size) {
            challenges[number].size = challenge.size - size;
        } else {
            require(size == challenge.size);
            delete challenges[number];
        }
    }

    /**
     * Returns 'amount' of the collateral to the challenger and reduces or deletes the relevant challenge.
     */
    function _returnChallengerCollateral(
        Challenge memory challenge,
        uint32 number,
        uint256 amount,
        bool postpone
    ) internal {
        _returnCollateral(challenge.position.collateral(), challenge.challenger, amount, postpone);
        if (challenge.size == amount) {
            // bid on full amount
            delete challenges[number];
        } else {
            // bid on partial amount
            challenges[number].size -= amount;
        }
    }

    /**
     * Calculates the current Dutch auction price.
     * @dev Starts at the full price at time 'start' and linearly goes to 0 as 'phase2' passes.
     */
    function _calculatePrice(uint64 start, uint64 phase2, uint256 liqPrice) internal view returns (uint256) {
        uint64 timeNow = uint64(block.timestamp);
        if (timeNow <= start) {
            return liqPrice;
        } else if (timeNow >= start + phase2) {
            return 0;
        } else {
            uint256 timeLeft = phase2 - (timeNow - start);
            return (liqPrice / phase2) * timeLeft;
        }
    }

    /**
     * Get the price per unit of the collateral for the given challenge.
     * @dev The price comes with (36-collateral.decimals()) digits, such that multiplying it with the
     * raw collateral amount always yields a price with 36 digits, or 18 digits after dividing by 10**18 again.
     */
    function price(uint32 challengeNumber) public view returns (uint256) {
        Challenge memory challenge = challenges[challengeNumber];
        if (challenge.challenger == address(0x0)) {
            return 0;
        } else {
            (uint256 liqPrice, uint64 phase1, uint64 phase2) = challenge.position.challengeData();
            return _calculatePrice(challenge.start + phase1, phase2, liqPrice);
        }
    }

    /**
     * Challengers can call this method to withdraw collateral whose return was postponed.
     */
    function returnPostponedCollateral(address collateral, address target) external {
        uint256 amount = pendingReturns[collateral][msg.sender];
        delete pendingReturns[collateral][msg.sender];
        IERC20(collateral).transfer(target, amount);
    }

    function _returnCollateral(IERC20 collateral, address recipient, uint256 amount, bool postpone) internal {
        if (postpone) {
            // Postponing helps in case the challenger was blacklisted or otherwise cannot receive at the moment.
            pendingReturns[address(collateral)][recipient] += amount;
            emit PostPonedReturn(address(collateral), recipient, amount);
        } else {
            collateral.transfer(recipient, amount); // return the challenger's collateral
        }
    }
}
```

2.G Position Factory

The position factory is a helper contract to create and clone positions.

```solidity
pragma solidity ^0.8.0;

import "./Position.sol";
import "./interface/IFrankencoin.sol";

contract PositionFactory {
    /**
     * Create a completely new position in a newly deployed contract.
     * Must be called through minting hub to be recognized as valid position.
     */
    function createNewPosition(
        address _owner,
        address _zchf,
        address _collateral,
        uint256 _minCollateral,
        uint256 _initialLimit,
        uint256 _initPeriod,
        uint256 _duration,
        uint64 _challengePeriod,
        uint32 _annualInterestPPM,
        uint256 _liqPrice,
        uint32 _reserve
    ) external returns (address) {
        return
            address(
                new Position(
                    _owner,
                    msg.sender,
                    _zchf,
                    _collateral,
                    _minCollateral,
                    _initialLimit,
                    _initPeriod,
                    _duration,
                    _challengePeriod,
                    _annualInterestPPM,
                    _liqPrice,
                    _reserve
                )
            );
    }

    /**
     * clone an existing position. This can be a clone of another clone,
     * or an original position.
     * @param _existing address of the position we want to clone
     * @return address of the newly created clone position
     */
    function clonePosition(address _existing) external returns (address) {
        Position existing = Position(_existing);
        Position clone = Position(_createClone(existing.original()));
        return address(clone);
    }

    // github.com/optionality/clone-factory/blob/32782f82dfc5a00d103a7e61a17a5dedbd1e8e9d/contracts/CloneFactory.sol
    function _createClone(address target) internal returns (address result) {
        bytes20 targetBytes = bytes20(target);
        assembly {
            let clone := mload(0x40)
            mstore(clone, 0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
            mstore(add(clone, 0x14), targetBytes)
            mstore(add(clone, 0x28), 0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
            result := create(0, clone, 0x37)
        }
        require(result != address(0), "ERC1167: create failed");
    }
}
```

2.H Collateralized Position

This smart contract represents a collateralized position. There is a separate instance of this contract for each position and each instance belongs to an owner. The owner is the person that created (or cloned) the position and also considered the owner of the provided collateral.

```
pragma solidity ^0.8.0;

import "./utils/Ownable.sol";
import "./utils/MathUtil.sol";

import "./interface/IERC20.sol";
import "./interface/IPosition.sol";
import "./interface/IReserve.sol";
import "./interface/IFrankencoin.sol";

/**
 * @title Position
 * A collateralized minting position.
 */
contract Position is Ownable, IPosition, MathUtil {
    /**
     * Note that this contract is intended to be cloned. All clones will share the same values for
     * the constant and immutable fields, but have their own values for the other fields.
     */

    /**
     * The zchf price per unit of the collateral below which challenges succeed, (36 - collateral.decimals) decimals
     */
    uint256 public price;

    /**
     * Net minted amount, including reserve.
     */
    uint256 public minted;

    /**
     * Amount of the collateral that is currently under a challenge.
     * Used to figure out whether there are pending challenges.
     */
    uint256 public challengedAmount;

    /**
     * Challenge period in seconds.
     */
    uint64 public immutable challengePeriod;

    /**
     * End of the latest cooldown. If this is in the future, minting is suspended.
     */
    uint256 public cooldown;

    /**
     * How much can be minted at most.
     */
    uint256 public limit;

    /**
     * Timestamp when minting can start and the position no longer denied.
     */
    uint256 public immutable start;

    /**
     * Timestamp of the expiration of the position. After expiration, challenges cannot be averted
     * any more. This is also the basis for fee calculations.
     */
    uint256 public expiration;

    /**
     * The original position to help identifying clones.
     */
    address public immutable original;

    /**
     * Pointer to the minting hub.
     */
    address public immutable hub;

    /**
     * The Frankencoin contract.
     */
    IFrankencoin public immutable zchf;

    /**
     * The collateral token.
     */
    IERC20 public immutable override collateral;

    /**
     * Minimum acceptable collateral amount to prevent dust.
     */
    uint256 public immutable override minimumCollateral;

    /**
     * Always pay interest for at least four weeks.
     */
    uint256 private constant MIN_INTEREST_DURATION = 4 weeks;

    /**
     * The interest in parts per million per year that is deducted when minting Frankencoins.
     * To be paid upfront.
     */
    uint32 public immutable annualInterestPPM;

    /**
     * The reserve contribution in parts per million of the minted amount.
     */
    uint32 public immutable reserveContribution;

    event MintingUpdate(uint256 collateral, uint256 price, uint256 minted, uint256 limit);
    event PositionDenied(address indexed sender, string message); // emitted if closed by governance

    error InsufficientCollateral();
    error TooLate();
    error RepaidTooMuch(uint256 excess);
    error LimitExceeded();
    error ChallengeTooSmall();
    error Expired();
    error Hot();
    error Challenged();
    error NotHub();

    modifier alive() {
        if (block.timestamp >= expiration) revert Expired();
        _;
    }

    modifier noCooldown() {
        if (block.timestamp <= cooldown) revert Hot();
        _;
    }

    modifier noChallenge() {
        if (challengedAmount > 0) revert Challenged();
        _;
    }

    modifier onlyHub() {
        if (msg.sender != address(hub)) revert NotHub();
        _;
    }

    /**
     * @dev See MintingHub.openPosition
     */
    constructor(
        address _owner,
        address _hub,
        address _zchf,
        address _collateral,
        uint256 _minCollateral,
        uint256 _initialLimit,
        uint256 _initPeriod,
        uint256 _duration,
        uint64 _challengePeriod,
        uint32 _annualInterestPPM,
        uint256 _liqPrice,
        uint32 _reservePPM
    ) {
        require(_initPeriod >= 3 days); // must be at least three days, recommended to use higher values
        _setOwner(_owner);
        original = address(this);
        hub = _hub;
        zchf = IFrankencoin(_zchf);
        collateral = IERC20(_collateral);
        annualInterestPPM = _annualInterestPPM;
        reserveContribution = _reservePPM;
        minimumCollateral = _minCollateral;
        challengePeriod = _challengePeriod;
        start = block.timestamp + _initPeriod; // at least three days time to deny the position
        cooldown = start;
        expiration = start + _duration;
        limit = _initialLimit;
        _setPrice(_liqPrice);
    }

    /**
     * Method to initialize a freshly created clone. It is the responsibility of the creator to make sure this is only
     * called once and to call reduceLimitForClone on the original position before initializing the clone.
     */
    function initializeClone(
        address owner,
        uint256 _price,
        uint256 _coll,
        uint256 _initialMint,
        uint256 expirationTime
    ) external onlyHub {
        if (_coll < minimumCollateral) revert InsufficientCollateral();
        uint256 impliedPrice = (_initialMint * ONE_DEC18) / _coll;
        _initialMint = (impliedPrice * _coll) / ONE_DEC18; // to cancel potential rounding errors
        if (impliedPrice > _price) revert InsufficientCollateral();
        _setOwner(owner);
        limit = _initialMint;
        expiration = expirationTime;
        _setPrice(impliedPrice);
        _mint(owner, _initialMint, _coll);
    }

    function limitForClones() public view returns (uint256) {
        uint256 backedLimit = (_collateralBalance() * price) / ONE_DEC18;
        if (backedLimit >= limit) {
            return 0;
        } else {
            // due to invariants, this is always below (limit - minted)
            return limit - backedLimit;
        }
    }

    /**
     * Adjust this position's limit to allow a clone to mint its own Frankencoins.
     * Invariant: global limit stays the same.
     *
     * Cloning a position is only allowed if the position is not challenged, not expired and not in cooldown.
     */
    function reduceLimitForClone(uint256 mint_) external noChallenge noCooldown alive onlyHub {
        if (mint_ > limitForClones()) revert LimitExceeded();
        limit -= mint_;
    }

    /**
     * Qualified pool share holders can call this method to immediately expire a freshly proposed position.
     */
    function deny(address[] calldata helpers, string calldata message) external {
        if (block.timestamp >= start) revert TooLate();
        IReserve(zchf.reserve()).checkQualified(msg.sender, helpers);
        _close(); // since expiration is immutable, we put it under eternal cooldown
        emit PositionDenied(msg.sender, message);
    }

    function _close() internal {
        cooldown = type(uint256).max;
    }

    function isClosed() public view returns (bool) {
        return cooldown == type(uint256).max;
    }

    /**
     * This is how much the minter can actually use when minting ZCHF, with the rest being used
     * assigned to the minter reserve or (if applicable) fees.
     */
    function getUsableMint(uint256 totalMint, bool afterFees) external view returns (uint256) {
        if (afterFees) {
            return (totalMint * (1000_000 - reserveContribution - calculateCurrentFee())) / 1000_000;
        } else {
            return (totalMint * (1000_000 - reserveContribution)) / 1000_000;
        }
    }

    /**
     * "All in one" function to adjust the outstanding amount of ZCHF, the collateral amount,
     * and the price in one transaction.
     */
    function adjust(uint256 newMinted, uint256 newCollateral, uint256 newPrice) external onlyOwner {
        uint256 colbal = _collateralBalance();
        if (newCollateral > colbal) {
            collateral.transferFrom(msg.sender, address(this), newCollateral - colbal);
        }
        // Must be called after collateral deposit, but before withdrawal
        if (newMinted < minted) {
            zchf.burnFromWithReserve(msg.sender, minted - newMinted, reserveContribution);
            minted = newMinted;
        }
        if (newCollateral < colbal) {
            withdrawCollateral(msg.sender, colbal - newCollateral);
        }
        // Must be called after collateral withdrawal
        if (newMinted > minted) {
            mint(msg.sender, newMinted - minted);
        }
        if (newPrice != price) {
            adjustPrice(newPrice);
        }
    }

    /**
     * Allows the position owner to adjust the liquidation price as long as there is no pending challenge.
     * Lowering the liquidation price can be done with immediate effect, given that there is enough collateral.
     * Increasing the liquidation price triggers a cooldown period of 3 days, during which minting is suspended.
     */
    function adjustPrice(uint256 newPrice) public onlyOwner noChallenge {
        if (newPrice > price) {
            _restrictMinting(3 days);
        } else {
            _checkCollateral(_collateralBalance(), newPrice);
        }
        _setPrice(newPrice);
        emit MintingUpdate(_collateralBalance(), price, minted, limit);
    }

    function _setPrice(uint256 newPrice) internal {
        require(newPrice * minimumCollateral <= limit * ONE_DEC18); // sanity check
        price = newPrice;
    }

    function _collateralBalance() internal view returns (uint256) {
        return IERC20(collateral).balanceOf(address(this));
    }

    /**
     * Mint ZCHF as long as there is no open challenge, the position is not subject to a cooldown,
     * and there is sufficient collateral.
     */
    function mint(address target, uint256 amount) public onlyOwner noChallenge noCooldown alive {
        _mint(target, amount, _collateralBalance());
    }

    function calculateCurrentFee() public view returns (uint32) {
        uint256 exp = expiration;
        uint256 time = block.timestamp < start ? start : block.timestamp;
        uint256 timePassed = time >= exp - MIN_INTEREST_DURATION ? MIN_INTEREST_DURATION : exp - time;
        // Time resolution is in the range of minutes for typical interest rates.
        return uint32((timePassed * annualInterestPPM) / 365 days);
    }

    function _mint(address target, uint256 amount, uint256 collateral_) internal {
        if (minted + amount > limit) revert LimitExceeded();
        zchf.mintWithReserve(target, amount, reserveContribution, calculateCurrentFee());
        minted += amount;

        _checkCollateral(collateral_, price);
        emit MintingUpdate(_collateralBalance(), price, minted, limit);
    }

    function _restrictMinting(uint256 period) internal {
        uint256 horizon = block.timestamp + period;
        if (horizon > cooldown) {
            cooldown = horizon;
        }
    }

    /**
     * Repay some ZCHF. If too much is repaid, the call fails.
     * It is possible to repay while there are challenges, but the collateral is locked until all is clear again.
     *
     * The repaid amount should fulfill the following equation in order to close the position,
     * i.e. bring the minted amount to 0:
     * minted = amount + zchf.calculateAssignedReserve(amount, reservePPM)
     *
     * Under normal circumstances, this implies:
     * amount = minted * (1000000 - reservePPM)
     *
     * E.g. if minted is 50 and reservePPM is 200000, it is necessary to repay 40 to be able to close the position.
     */
    function repay(uint256 amount) public {
        IERC20(zchf).transferFrom(msg.sender, address(this), amount);
        uint256 actuallyRepaid = IFrankencoin(zchf).burnWithReserve(amount, reserveContribution);
        _notifyRepaid(actuallyRepaid);
        emit MintingUpdate(_collateralBalance(), price, minted, limit);
    }

    function _notifyRepaid(uint256 amount) internal {
        if (amount > minted) revert RepaidTooMuch(amount - minted);
        minted -= amount;
    }

    /**
     * Withdraw any ERC20 token that might have ended up on this address.
     * Withdrawing collateral is subject to the same restrictions as withdrawCollateral(...).
     */
    function withdraw(address token, address target, uint256 amount) external onlyOwner {
        if (token == address(collateral)) {
            withdrawCollateral(target, amount);
        } else {
            uint256 balance = _collateralBalance();
            IERC20(token).transfer(target, amount);
            require(balance == _collateralBalance()); // guard against double-entry-point tokens
        }
    }

    /**
     * Withdraw collateral from the position up to the extent that it is still well collateralized afterwards.
     * Not possible as long as there is an open challenge or the contract is subject to a cooldown.
     *
     * Withdrawing collateral below the minimum collateral amount formally closes the position.
     */
    function withdrawCollateral(address target, uint256 amount) public onlyOwner noChallenge {
        if (block.timestamp <= cooldown && !isClosed()) revert Hot();
        uint256 balance = _withdrawCollateral(target, amount);
        _checkCollateral(balance, price);
        if (balance < minimumCollateral && balance > 0) revert InsufficientCollateral(); // Prevent dust amounts
    }

    function _withdrawCollateral(address target, uint256 amount) internal returns (uint256) {
        if (amount > 0) {
            // Some weird tokens fail when trying to transfer 0 amounts
            IERC20(collateral).transfer(target, amount);
        }
        uint256 balance = _collateralBalance();
        if (balance < minimumCollateral && challengedAmount == 0) {
            // This leaves a slightly unsatisfying possibility open: if the withdrawal happens due to a successful
            // challenge, there might be a small amount of collateral left that is not withheld in case there are no
            // other pending challenges. The only way to cleanly solve this would be to have two distinct cooldowns,
            // one for minting and one for withdrawals.
            _close();
        }

        emit MintingUpdate(balance, price, minted, limit);
        return balance;
    }

    /**
     * This invariant must always hold and must always be checked when any of the three
     * variables change in an adverse way.
     */
    function _checkCollateral(uint256 collateralReserve, uint256 atPrice) internal view {
        if (collateralReserve * atPrice < minted * ONE_DEC18) revert InsufficientCollateral();
    }

    /**
     * Returns the liquidation price and the durations for phase1 and phase2 of the challenge.
     * In this implementation, both phases are always of equal length.
     */
    function challengeData() external view returns (uint256 liqPrice, uint64 phase1, uint64 phase2) {
        return (price, challengePeriod, challengePeriod);
    }

    function notifyChallengeStarted(uint256 size) external onlyHub {
        // Require minimum size. Collateral balance can be below minimum if it was partially challenged before.
        if (size < minimumCollateral && size < _collateralBalance()) revert ChallengeTooSmall();
        if (size == 0) revert ChallengeTooSmall();
        challengedAmount += size;
    }

    /**
     * @param size   amount of collateral challenged (dec18)
     */
    function notifyChallengeAverted(uint256 size) external onlyHub {
        challengedAmount -= size;
        // Don't allow minter to close the position immediately so challenge can be repeated before
        // the owner has a chance to mint more on an undercollateralized position
        _restrictMinting(1 days);
    }

    /**
     * Notifies the position that a challenge was successful.
     * Triggers the payout of the challenged part of the collateral.
     * Everything else is assumed to be handled by the hub.
     *
     * @param _bidder   address of the bidder that receives the collateral
     * @param _size     size of the collateral bid for (dec 18)
     * @return (position owner, effective challenge size in ZCHF, repaid amount, reserve ppm)
     */
    function notifyChallengeSucceeded(
        address _bidder,
        uint256 _size
    ) external onlyHub returns (address, uint256, uint256, uint32) {
        challengedAmount -= _size;
        uint256 colBal = _collateralBalance();
        if (colBal < _size) {
            _size = colBal;
        }
        uint256 repayment = _mulDiv(minted, _size, colBal);
        _notifyRepaid(repayment); // we assume the caller takes care of the actual repayment
        _withdrawCollateral(_bidder, _size); // transfer collateral to the bidder and emit update

        // Give time for additional challenges before the owner can mint again. In particular,
        // the owner might have added collateral only seconds before the challenge ended, preventing a close.
        _restrictMinting(3 days);

        return (owner, _size, repayment, reserveContribution);
    }
}
```

2.I Math Utilities

A contract with some basic mathematical utilities. Most notable is a function to calculate the cubic root using Halley's approximation method as Solidity only supports integer exponents natively.

```
pragma solidity ^0.8.0;

/**
 * @title Functions for share valuation
 */
contract MathUtil {
    uint256 internal constant ONE_DEC18 = 10 ** 18;

    // Let's go for 12 digits of precision (18-6)
    uint256 internal constant THRESH_DEC18 = 10 ** 6;

    /**
     * Cubic root with Halley approximation
     *         Number 1e18 decimal
     * @param _v     number for which we calculate x**(1/3)
     * @return returns _v**(1/3)
     */
    function _cubicRoot(uint256 _v) internal pure returns (uint256) {
        // Good first guess for _v slightly above 1.0, which is often the case in the Frankencoin system
        uint256 x = _v > ONE_DEC18 && _v < 10 ** 19 ? (_v - ONE_DEC18) / 3 + ONE_DEC18 : ONE_DEC18;
        uint256 diff;
        do {
            uint256 powX3 = _mulD18(_mulD18(x, x), x);
            uint256 xnew = _mulDiv(x, (powX3 + 2 * _v), (2 * powX3 + _v));
            diff = xnew > x ? xnew - x : x - xnew;
            x = xnew;
        } while (diff > THRESH_DEC18);
        return x;
    }

    /**
     * Divides and multiplies, with divisor > 0.
     */
    function _mulDiv(uint256 x, uint256 factor, uint256 divisor) internal pure returns (uint256) {
        if (factor == 0) {
            return 0;
        } else if (type(uint256).max / factor > x) {
            return (x * factor) / divisor;
        } else {
            // divide first to avoid overflow
            return x > factor ? (x / divisor) * factor : (factor / divisor) * x;
        }
    }

    function _mulD18(uint256 _a, uint256 _b) internal pure returns (uint256) {
        return (_a * _b) / ONE_DEC18;
    }

    function _divD18(uint256 _a, uint256 _b) internal pure returns (uint256) {
        return (_a * ONE_DEC18) / _b;
    }

    function _power3(uint256 _x) internal pure returns (uint256) {
        return _mulD18(_mulD18(_x, _x), _x);
    }

    function _min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }
}
```

2.J Stablecoin Bridge

A minter contract that allows the minting of Frankencoins using a trusted stablecoin with the same reference currency. Separate instances of bridges are required to support multiple bridged coins or to mint beyond the specified limit.

```
\begin{spacing}{0.5}
pragma solidity ^0.8.0;

import "./interface/IERC20.sol";
import "./interface/IERC677Receiver.sol";
import "./interface/IFrankencoin.sol";

/**
 * @title Stable Coin Bridge
 * A minting contract for another Swiss franc stablecoin ('source stablecoin') that we trust.
 * @author Frankencoin
 */
contract StablecoinBridge {
    IERC20 public immutable chf; // the source stablecoin
    IFrankencoin public immutable zchf; // the Frankencoin

    /**
     * The time horizon after which this bridge expires and needs to be replaced by a new contract.
     */
    uint256 public immutable horizon;

    /**
     * The maximum amount of outstanding converted source stablecoins.
     */
    uint256 public immutable limit;
    uint256 public minted;

    error Limit(uint256 amount, uint256 limit);
    error Expired(uint256 time, uint256 expiration);
    error UnsupportedToken(address token);

    constructor(address other, address zchfAddress, uint256 limit_) {
        chf = IERC20(other);
        zchf = IFrankencoin(zchfAddress);
        horizon = block.timestamp + 52 weeks;
        limit = limit_;
        minted = 0;
    }

    /**
     * Convenience method for mint(msg.sender, amount)
     */
    function mint(uint256 amount) external {
        mintTo(msg.sender, amount);
    }

    /**
     * Mint the target amount of Frankencoins, taking the equal amount of source coins from the sender.
     * @dev This only works if an allowance for the source coins has been set and the caller has enough of them.
     */
    function mintTo(address target, uint256 amount) public {
        chf.transferFrom(msg.sender, address(this), amount);
        _mint(target, amount);
    }

    function _mint(address target, uint256 amount) internal {
        if (block.timestamp > horizon) revert Expired(block.timestamp, horizon);
        zchf.mint(target, amount);
        minted += amount;
        if (minted > limit) revert Limit(amount, limit);
    }

    /**
     * Convenience method for burnAndSend(msg.sender, amount)
     */
    function burn(uint256 amount) external {
        _burn(msg.sender, msg.sender, amount);
    }

    /**
     * Burn the indicated amount of Frankencoin and send the same number of source coin to the caller.
     */
    function burnAndSend(address target, uint256 amount) external {
        _burn(msg.sender, target, amount);
    }

    function _burn(address zchfHolder, address target, uint256 amount) internal {
        zchf.burnFrom(zchfHolder, amount);
        chf.transfer(target, amount);
        minted -= amount;
    }
}
\end{spacing}
```

References

Acharya, Viral V et al. (2010). *Regulating Wall Street: The Dodd-Frank Act and the new architecture of global finance*. Vol. 608. Wiley Hoboken, NJ.

Adams, Hayden et al. (2021). *Uniswap v3 Core*. URL: uniswap.org/whitepaper-v3.pdf.

Basel Committee on Banking Supervision (2010). “Basel III: A global regulatory framework for more resilient banks and banking systems”. In: *Bank For International Settlements*.

– (Dec. 2017). *High-level summary of Basel III reforms*. bis.org/bcbs/publ/d424_hlsummary.pdf.

– (Dec. 2019a). *CRE Calculation of RWA for Credit Risk*. bis.org/basel_framework/chapter/CRE/22.htm.

– (Dec. 2019b). *RBC Risk-based capital requirements*. bis.org/basel_framework/chapter/RBC/20.htm.

– (2022). *Prudential treatment of cryptoasset exposures*. URL: bis.org/bcbs/publ/d545.htm.

Beck, Roman, Christoph Müller-Bloch, and John Leslie King (2018). “Governance in the blockchain economy: A framework and research agenda”. In: *Journal of the association for information systems* 19.10, p. 1.

Bell, Frederick W and Neil B Murphy (1968). “Economies of scale and division of labor in commercial banking”. In: *Southern Economic Journal*, pp. 131–139.

Benston, George J, Gerald A Hanweck, and David B Humphrey (1982). “Scale economies in banking: A restructuring and reassessment”. In: *Journal of money, credit and banking* 14.4, pp. 435–456. DOI: 10.2307/1991654.

Björk, Tomas (2009). *Arbitrage theory in continuous time*. Oxford university press.

Breeden, Douglas T and Robert H Litzenberger (1978). “Prices of state-contingent claims implicit in option prices”. In: *Journal of business*, pp. 621–651.

Briola, Antonio et al. (2023). “Anatomy of a Stablecoin’s failure: The Terra-Luna case”. In: *Finance Research Letters* 51, p. 103358.

Brunnermeier, Markus K and Dirk Niepelt (2019). “On the equivalence of private and public money”. In: *Journal of Monetary Economics* 106, pp. 27–41.

Chaum, David, Christian Grothoff, and Thomas Moser (2021). “How to issue a central bank digital currency”. In: *SNB Working Papers*. URL: snb.ch/en/mmr/papers/id/working_paper_2021_03.

Chen, Jason, Kathy Fogel, and Kose John (2022). “Understanding the Maker Protocol”. In: *arXiv preprint arXiv:2210.16899*.

Christensen, Rune et al. (2021). *MKR Governance 101*. URL: makerdao.com/governance.

Clark, Jeffrey A (1984). “Estimation of economies of scale in banking using a generalized functional form”. In: *Journal of Money, Credit and Banking* 16.1, pp. 53–68. DOI: 0.2307/1992648.

Clements, Ryan (2021). “Built to Fail: The Inherent Fragility of Algorithmic Stablecoins”. In: *Wake Forest L. Rev. Online* 11, p. 131.

code4rena (2023). *Frankencoin Findings & Analysis Report*. Tech. rep. code4rena. URL: code4rena.com/reports/2023-04-frankencoin.

Efron, Bradley (1992). “Bootstrap methods: another look at the jackknife”. In: *Breakthroughs in statistics*. Springer, pp. 569–593.

Ellinger, Eleunthia Wong et al. (2020). “Skin in the Game: The Transformational Potential of Decentralized Autonomous Organizations”. In: 1. URL: researchgate.net/publication/371831972.

Foo, Chek Yang and Elina MI Koivisto (2004). “Defining grief play in MMORPGs: player and developer perceptions”. In: *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pp. 245–250.

Fritsch, Robin, Marino Müller, and Roger Wattenhofer (2022). “Analyzing voting power in decentralized governance: Who controls daos?” In: *arXiv preprint arXiv:2204.01176*.

Fu, Shange et al. (2023). “Rational Ponzi Game in Algorithmic Stablecoin”. In: *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, pp. 1–6.

Gilbert, R Alton (1984). “Bank market structure and competition: a survey”. In: *Journal of Money, Credit and Banking* 16.4, pp. 617–645. DOI: 10.2307/1992096.

Glosten, Lawrence R and Paul R Milgrom (1985). “Bid, ask and transaction prices in a specialist market with heterogeneously informed traders”. In: *Journal of financial economics* 14.1, pp. 71–100.

Ilo, Oecd (2015). “The labour share in G20 economies”. In: *Report prepared for the G20 Employment Working Group, Antalya, February*, p. 101.

Jarrow, Robert A and Stuart McLean Turnbull (2000). *Derivative securities.* South-Western Pub.

Jentzsch, Christoph (2016). *Decentralized autonomous organization to automate governance.* URL: lawofthelevel.lexblogplatformthree.com/wp-content/uploads/sites/187/2017/07/WhitePaper-1.pdf.

Jorion, Philippe et al. (2010). *Financial Risk Manager Handbook: FRM Part I/Part II.* Vol. 625. John Wiley & Sons.

Lauko, Robert and Richard Pardoe (2021). *Liquity: Decentralized Borrowing Protocol.* URL: docsend.com/view/bwiczmy.

Liu, Jiageng, Igor Makarov, and Antoinette Schoar (2023). *Anatomy of a Run: The Terra Luna Crash.* Tech. rep. National Bureau of Economic Research.

Mackinga, Torgin, Enis Ulqinaku, et al. (2023). *Code Assessment of the Frankencoin Smart Contracts.* Tech. rep. Chainsecurity.

Meisser, Luzius (2017). “The Code is the Model”. In: *International Journal of Microsimulation* 10.3, pp. 184–201.

Morrison, Robbie, Natasha CHL Mazey, and Stephen C Wingreen (2020). “The DAO controversy: the case for a new species of corporate governance?” In: *Frontiers in Blockchain* 3, p. 25.

Palfrey, Thomas R and Howard Rosenthal (1983). “A strategic calculus of voting”. In: *Public choice* 41.1, pp. 7–53.

Scherer, Mathias (2023). *Audit Report Frankencoin.* Tech. rep. Blockbite. URL: github.com/Frankencoin-ZCHF/FrankenCoin/blob/main/audits/blockbite-audit.pdf.

Vogelsteller, Fabian and Vitalik Buterin (2015). *ERC-20: Token Standard.* URL: eips.ethereum.org/EIPS/eip-20.

Wheatley, Martin (2012). “The Wheatley review of LIBOR”. In: *Final report.*