XPower Banq

A Permissionless DeFi Lending Protocol

Protocol, Engineering Primitives, Theory, Evaluation, and Reference

Karun The Ritch*

preprint:2605 [q-fin.CP] 1 May 2026

*ktr@xpowerbanq.com https://www.xpowerbanq.com

Abstract

DeFi lending markets channel billions of dollars but share a structural weakness: a sharp price move triggers liquidations, the dumped collateral pushes prices lower, and a cascade follows. XPower Banq is a permissionless lending protocol designed to dampen these cascades — along with several other systemic risks — without giving up the openness and composability that make DeFi work.

This volume gathers the protocol's complete documentation in one bound document. It comprises six companion papers reproduced here without abridgement — five of them individually published as arXiv-style preprints — organised into five parts.

Part I — Protocol. The whitepaper introduces five design choices that set the protocol apart from incumbents such as Compound and Aave [8, 9]. (1) *Optionally locked positions:* borrowers may lock their position for a fixed term in exchange for liquidation immunity; the locked share ϕ attenuates cascade severity by a factor of $(1-\phi)$. (2) *Lethargic governance:* every parameter change is bounded multiplicatively ($0.5\times$–$2\times$ per cycle) and phased in asymptotically, so users always have time to react. (3) *Beta-distributed position caps* slow how fast a single account can scale up, yielding Sybil resistance that grows as $\sqrt{n+2}$. (4) *Transferable debt positions* with inverted ERC20 semantics let liquidators *assume* bad debt rather than repay it — a far more capital-efficient model. (5) *Log-space TWAP oracles* compute time-weighted geometric means in both directions, cheaply and without overflow.

Part II — Engineering Primitives. Two implementation papers describe the lower-level on-chain building blocks the protocol depends on. A 16-slot quarterly *ring-buffer time lock* [3] tracks total locked depth in $O(1)$ via the algebraic identity $D = \Sigma Q - Tt + pL$, avoiding the gas cost of iterating slots. A *log-space compounding interest index* [4] stores the cumulative sum $L = \sum r_i$ in place of the multiplicative product, eliminating overflow and replacing write-path exponentiation with addition.

Part III — Theory. Formal foundations and proofs covering continuous compounding, log-space oracle aggregation, time-weighted parameter integration, and token-bucket rate limiting; proofs that the protocol resists liquidation cascades, Sybil amplification, abrupt governance, and MEV front-running; and a Nash-equilibrium analysis of lock adoption that identifies two self-reinforcing equilibria — low and high — with utilization-dependent dynamics.

Part IV — Evaluation. Four empirical studies validate the protocol. *Capacity accumulation* under the beta cap confirms the predicted $O(\sqrt{n})$ Sybil-resistance scaling. *Liquidation-cascade simulations* across five market-impact regimes show up to 80% cascade reduction at full lock adoption. *Log-space TWAP oracles* are stress-tested under price shocks and liquidity changes across 60 on-chain Foundry scenarios, exhibiting two-tick manipulation immunity. *Bad-debt risk* is quantified by Merton jump-diffusion Monte Carlo and corroborated by a closed-form analytical bound: at the default 66.67% LTV bad debt remains bounded under empirical crash distributions, and the 33% conservative-mode floor keeps bad debt at zero for crashes up to 50%.

Part V — Reference. A consolidated glossary of over 120 terms — protocol mechanisms, mathematical notation, and DeFi terminology — intended to be read alongside the body of the volume.

The defaults strike a deliberate balance between capital efficiency and solvency: 66.67% LTV with a 50% over-collateralisation buffer, adjustable through lethargic governance (§I. 4.3).

Keywords: DeFi, lending protocol, liquidation, oracle, TWAP, governance, interest rates, time locks, log-space arithmetic, Sybil resistance, Nash equilibrium

Contents

I Protocol 5
I. Protocol Whitepaper 7
I. 1 Introduction 7
I. 2 Related Work 7
I. 3 Protocol Architecture 7
I. 4 Core Mechanisms 8
I. 4.1 Locked Positions and Cascade Attenuation 8
I. 4.2 Position Transfer Semantics 8
I. 4.3 Lethargic Governance 9
I. 4.4 Beta-Distributed Position Caps 9
I. 4.5 Health Factor and Liquidation 10
I. 4.6 Oracle TWAP 10
I. 4.7 Interest Rates 10
I. 5 Anti-Spam Protection 10
I. 6 Governance and Parameters 11
I. 7 Security Analysis 11
I. 7.1 Adversary Model 11
I. 7.2 Core Security Properties 11
I. 7.3 Risk Assessment 11
I. 8 Evaluation 11
I. 8.1 Agent-Based Cascade Simulation 11
I. 8.2 Gas Cost Analysis 12
I. 9 Limitations and Future Work 12
I. 10 Conclusion 13
I. 11 Protocol Parameters 13
I. 11.1 Pool Parameters 13
I. 11.2 Position Parameters 13
I. 11.3 Oracle Parameters 13
I. 11.4 Vault Parameters 13
I. 11.5 Change Rate Constraints 13
I. 11.6 Role-Based Access Control 14
I. 12 Protocol Constants 14
I. 13 EMA Decay Factors 14

II Engineering Primitives 15
IIa. Ring-Buffer Time Locks 17
IIa. 1 Introduction 17
IIa. 2 Related Work 17
IIa. 3 Preliminaries 17
IIa. 3.1 Definitions 17
IIa. 3.2 Storage Layout 17
IIa. 3.3 Bitmap Encoding 17
IIa. 4 Ring-Lock Mechanism 18
IIa. 4.1 `more(user, amount, dt_term)` — $O(1)$ Lock Addition 18
IIa. 4.2 `free(user)` — $O(k)$ Expired Slot Sweep 18
IIa. 4.3 `push(src, tgt, mul, div)` — $O(k)$ Proportional Transfer 18
IIa. 4.4 `stateAt(user, stamp)` — $O(k)$ Exact Query 19
IIa. 4.5 `roll(user, amount, dt_term)` — $O(k)$ Sweep-and-Re-Lock 19
IIa. 4.6 `totalOf(user)` — $O(1)$ Cached Query 19
IIa. 5 Time-Lock Extension 19
IIa. 5.1 The Token-Seconds Integral 19
IIa. 5.2 The $O(1)$ Depth Identity 20
IIa. 5.3 `more()` with Depth Tracking 20
IIa. 5.4 `free()` with Depth Tracking 20
IIa. 5.5 `push()` with Depth Tracking 20
IIa. 5.6 `stateAt()` with Depth 21
IIa. 5.7 `depthOf(user)` — $O(1)$ Cached Depth 21
IIa. 5.8 Worked Example 21
IIa. 6 Invariants and Correctness Proofs 21
IIa. 6.1 Collision Freedom 21
IIa. 6.2 Depth Identity Correctness 22
IIa. 6.3 Cached vs Exact Invariants 22
IIa. 6.4 Self-Healing Correctness 22
IIa. 6.5 Depth Conservation Under Push 22
IIa. 6.6 Stale-Slot Corruption Without Free 23
IIa. 6.7 Non-Negativity 23
IIa. 6.8 Bitmap Consistency 23
IIa. 6.9 Permanent Lock Invariance 23
IIa. 7 Gas Analysis 23
IIa. 8 Integration 23
IIa. 8.1 Position Layer 23
IIa. 8.2 Graduated Lock Bonus/Malus 24
IIa. 9 Conclusion 24
IIb. Log-Space Compounding Index 27
IIb. 1 Introduction 27
IIb. 2 Related Work 27
IIb. 3 Overflow Analysis 27
IIb. 3.1 Budget Computation 27
IIb. 3.2 Time-to-Overflow 27
IIb. 3.3 Boundedness of Other Values 28
IIb. 4 Log-Space Index 28
IIb. 4.1 Core Insight 28
IIb. 4.2 Formal Definition 28
IIb. 4.3 Storage Transformation 28
IIb. 4.4 Shipped Storage Layout 28
IIb. 4.5 Overflow Elimination 29
IIb. 4.6 Invariants 29
IIb. 5 Code Transformation 29
IIb. 5.1 Accrual: `_indexOf` 29
IIb. 5.2 Balance Query: `totalOf` 29
IIb. 5.3 Lock Yield: `_lockYieldOf` 30
IIb. 5.4 Per-User Snapshots 30
IIb. 6 Gas Analysis 30
IIb. 6.1 Theoretical Cost Model 30
IIb. 6.2 Empirical Benchmarks 31
IIb. 7 Precision Analysis 31
IIb. 7.1 Theoretical Bound 31
IIb. 7.2 Empirical Measurement 31
IIb. 7.3 Root Cause 31
IIb. 8 Adversarial Analysis 32
IIb. 8.1 Rounding Manipulation 32
IIb. 8.2 Dust Extraction via Read-Time Rounding 32
IIb. 8.3 Gas Griefing on Read Path 32
IIb. 8.4 Timestamp Sensitivity and Summary 32
IIb. 9 Limitations and Future Work 32
IIb. 9.1 Future Work 32
IIb. 10 Conclusion 32

III Theory 35
III. Mathematical Theory & Proofs 37

III. 1 **Mathematical Foundations** 37
III. 1.1 Interest Accrual . . . 37
III. 1.2 Time-Weighted Integration . . . 37
III. 1.3 Oracle Price Aggregation . . . 37
III. 1.4 Rate Limiting . . . 37
III. 2 **Formal Proofs** 37
III. 2.1 Cascade Attenuation Proof . . . 37
III. 2.2 Sybil Rate-Limiting . . . 37
III. 2.3 Governance Rate Bound Proof . . . 38
III. 2.4 MEV Front-Running Resistance . . . 38
III. 3 **Nash Equilibrium Analysis for Lock Adoption** 38
III. 3.1 Model Setup . . . 38
III. 3.1.1 Position Types and Rates . . . 38
III. 3.1.2 Secondary Market Dynamics . . . 38
III. 3.2 Player Utility Functions . . . 39
III. 3.2.1 Supplier Utility . . . 39
III. 3.2.2 Breakeven Condition . . . 39
III. 3.3 Breakeven Analysis by Utilization . . . 39
III. 3.4 Liquidation Seniority Value . . . 39
III. 3.5 Nash Equilibrium Characterization . . . 39
III. 3.5.1 Equilibrium Condition . . . 39
III. 3.5.2 Utilization-Dependent Equilibria . . . 39
III. 3.6 Protocol Margin Analysis . . . 39
III. 3.6.1 Margin as Function of Lock Adoption 39
III. 3.6.2 Solvency at Full Adoption . . . 40
III. 3.7 Stability Analysis . . . 40
III. 3.7.1 Multiple Equilibria . . . 40
III. 3.7.2 Equilibrium Selection . . . 40
III. 3.8 Summary . . . 40
III. 3.8.1 Equilibrium Structure . . . 40
III. 3.8.2 Cascade Dynamics and Keeper Sizing 40
III. 3.8.3 Solvency Guarantees . . . 41
III. 3.8.4 Welfare . . . 41

IV **Evaluation** 43
IV. **Simulations & Risk Analysis** 45
IV. 1 **Capacity Accumulation Simulation** 45
IV. 1.1 Cap Function Components . . . 45
IV. 1.2 Simulation Algorithm . . . 45
IV. 1.3 Convergence Behavior . . . 45
IV. 1.4 Representative Output . . . 45
IV. 1.5 Multi-Account Share Dynamics . . . 45
IV. 1.6 Limitations . . . 46
IV. 2 **Cascade Simulation Implementation** 46
IV. 2.1 Multi-Scenario Comparison . . . 46
IV. 2.2 Detailed Per-Scenario Analysis . . . 46
IV. 2.3 Algorithm . . . 47
IV. 2.4 Market Impact Model . . . 47
IV. 2.5 Limitations . . . 47
IV. 3 **TWAP Oracle Simulations** 47
IV. 3.1 Decay Factor Visualization . . . 47
IV. 3.2 Price Shock Response . . . 47
IV. 3.3 Spread Scaling by Liquidity . . . 48
IV. 3.4 Complete Spread Analysis . . . 48
IV. 3.5 On-Chain Scenario Testing . . . 48
IV. 3.5.1 Token/Liquidity Configurations . . . 49
IV. 3.5.2 Mid-Price Manipulation Resistance . 49
IV. 3.5.3 Spread Auto-Widening Response . . 49
IV. 3.5.4 Decimal Invariance . . . 49
IV. 3.5.5 Strengths and Weaknesses . . . 49
IV. 4 **Bad Debt Risk Quantification** 50
IV. 4.1 Introduction . . . 50
IV. 4.2 Oracle Lag Model . . . 50
IV. 4.2.1 EMA Update Semantics . . . 50
IV. 4.2.2 Step-Crash Convergence . . . 50
IV. 4.2.3 Phantom-Healthy Windows . . . 51
IV. 4.2.4 Worst-Case Blind Time . . . 51
IV. 4.3 Monte Carlo Simulation . . . 51
IV. 4.3.1 Price Process . . . 51
IV. 4.3.2 Simulation Design . . . 51
IV. 4.3.3 Results . . . 52
IV. 4.3.4 Bad Debt Distribution . . . 52
IV. 4.3.5 Drawdown–Bad Debt Relationship . 52
IV. 4.3.6 Liquidation Delay Distribution . . . 52
IV. 4.4 Analytical Bound . . . 53
IV. 4.4.1 Instant Liquidation Baseline . . . 53
IV. 4.4.2 Oracle Delay Penalty . . . 53
IV. 4.4.3 Conservative Upper Bound . . . 53
IV. 4.5 Safe Operating Region . . . 53
IV. 4.5.1 Safety Criterion . . . 53
IV. 4.5.2 Safe Configurations . . . 53
IV. 4.5.3 Key Observations . . . 54
IV. 4.6 Sensitivity Analysis . . . 54
IV. 4.6.1 Parameter Rankings . . . 54
IV. 4.6.2 Interaction Effects . . . 55
IV. 4.7 Partial Liquidation . . . 55
IV. 4.8 Cross-Validation . . . 55
IV. 4.8.1 Oracle Model vs. Solidity . . . 55
IV. 4.8.2 Phantom Window Verification . . . 55
IV. 4.8.3 Bound Conservatism . . . 55
IV. 4.9 Deployment Recommendations . . . 55
IV. 4.10 Conclusion . . . 56

V **Reference** 57
V. **References & Glossary** 59
References 59
Glossary 61

XPower Banq: Part I

Protocol Whitepaper

A Permissionless DeFi Lending Protocol with Lethargic Governance and Beta-Distributed Position Caps

Abstract

We present **XPower Banq**, a permissionless DeFi lending protocol on the Ethereum Virtual Machine. The protocol introduces: (1) *optionally locked positions* that attenuate liquidation cascades by factor $(1-\phi)$, (2) *lethargic governance* with time-weighted parameter transitions bounded by $0.5\times$–$2\times$ per cycle, (3) *beta-distributed position caps* with holder-count scaling that rate-limits capacity accumulation to $O(\sqrt{k})$ for k accounts, (4) *transferable debt positions* with inverted ERC20 semantics enabling capital-efficient debt assumption liquidation, and (5) *log-space TWAP oracles* with bidirectional geometric mean spread computation and logarithmic spread scaling. The protocol balances capital efficiency against bounded bad-debt risk: default parameters yield 66.67% LTV with a 50% over-collateralization buffer, adjustable via lethargic governance (§I. 4.3). We provide technical analysis of the architecture, security properties, simulation results, and limitations.

Keywords: DeFi, Lending Protocol, Liquidation, Oracle, TWAP, Governance, Interest Rates

I. 1 Introduction

Decentralized lending protocols—Compound [8], Aave [9], MakerDAO [10]—have established algorithmic interest rate markets as foundational DeFi infrastructure [14]. Persistent challenges remain: oracle manipulation [18], governance attacks [26], capital inefficiency, and systemic liquidation cascades [16].

XPower Banq addresses these through five design choices:

1. **Locked Positions**: Irrevocable position locks attenuate liquidation cascades by factor $(1-\phi)$ through preventing immediate collateral redemption (Section I. 4.1).

2. **Lethargic Governance**: Multiplicative bounds ($0.5\times$–$2\times$) with asymptotic transitions prevent governance shocks (Section I. 4.3).

3. **Beta-Distributed Caps**: The $12\lambda(1-\lambda)^2/\sqrt{n+2}$ cap function rate-limits capacity accumulation sublinearly (Section I. 4.4).

4. **Debt Assumption Liquidation**: Transferable debt positions with inverted ERC20 semantics eliminate liquidator capital requirements (Section I. 4.5).

5. **Log-Space TWAP Oracle**: Geometric mean temporal averaging with bidirectional spread computation resists flash loan manipulation (Section I. 4.6).

The remainder of this paper is organized as follows. Section IIb. 2 surveys related work and positions our contributions against existing protocols. Section I. 3 presents the contract architecture, followed by Section I. 4 which details the core mechanisms. Section I. 5 describes anti-spam protection, and Section I. 6 covers the governance model. Section I. 7 provides a security analysis under an explicit adversary model. Section I. 8 presents simulation and gas evaluation results, Section IIb. 9 discusses limitations, and Section IIb. 10 concludes. Appendices A–C provide specifications and reference tables. A companion document [5, 6, 7] provides extended mathematical analysis, formal proofs, simulation implementations, and reference material.

I. 2 Related Work

Lending Protocols. Compound [8] introduced algorithmic interest rate markets with utilization-based rate curves and 2-day governance timelocks. Aave [9] extended this with flash loans, stable rate borrowing, and credit delegation; V3 introduced E-Mode for correlated assets, though liquidation still requires liquid capital. MakerDAO [10] pioneered the CDP model, suffering $8.3M in losses during "Black Thursday" [16]. Euler [11] pioneered permissionless asset listing with risk-tiered collateral and a reactive rate model. Liquity [12] introduced the Stability Pool where pre-deposited capital absorbs liquidations; XPower Banq instead transfers debt to liquidators. Morpho [13] optimizes rates via peer-to-peer matching atop existing protocols.

Liquidation and MEV. Gudgeon et al. [16] analyzed the March 2020 crisis in which cascade effects drained significant protocol value. Perez et al. [17] showed that liquidation markets are highly concentrated, with the top 5 liquidators capturing over 80% of value, and Daian et al. [23] and Qin et al. [24] quantified MEV extraction from liquidation events.

Oracles. Mackinga et al. [18] demonstrated practical TWAP manipulation, motivating several mitigation approaches. Chainlink [19] provides off-chain aggregation but carries staleness risks during network congestion [15]. Uniswap V3 [22] introduced geometric mean TWAP computed in ring buffers, and Angeris and Chitra [20] formalized manipulation cost bounds for constant function market makers.

Governance. Beanstalk's $182M governance attack [26] exploited same-block execution, underscoring the need for rate-bounded governance. Standard timelocks [38] constrain the timing of changes but not their magnitude. XPower Banq addresses this with lethargic governance that requires multiple cycles for any significant parameter change.

Comparative Analysis. Table 1 compares protocols across key dimensions. Some features marked absent may be deliberately omitted by design (e.g., Morpho Blue delegates oracle validation to curators), and XPower Banq's advantages carry trade-offs discussed in Section IIb. 9.

I. 3 Protocol Architecture

The protocol comprises six core contracts, depicted in Figure 1:

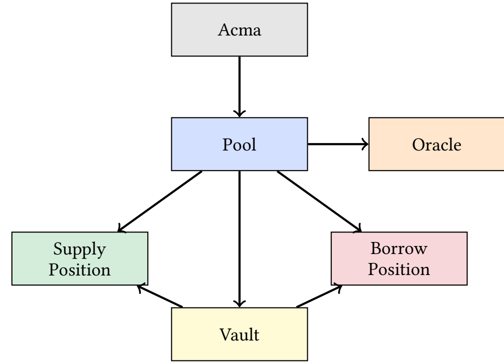


Figure 1: XPower Banq contract architecture

1. **Pool**: The main contract, managing supply, borrow, settle, and redeem operations with health checks and liquidation logic.

2. **Position**: ERC20 [45] tokens representing supply and borrow positions, each with distinct transfer semantics (Section I. 4.2).

3. **Vault**: An ERC4626-compliant [46] custody contract that tracks deposited assets and utilization.

4. **Oracle**: A log-space TWAP aggregator with bidirectional geomean spread computation.

Table 1: Comparative analysis of DeFi lending protocols. Entries reflect architectural choices; absence of a feature may be deliberate.

Feature	Compound	Aave	MakerDAO	Liquity	Euler	**XPower Banq**
1. Liquidation Model	Repayment	Repayment	Auction	Stability Pool	Repayment	Debt Assumption
2. Liquidator Capital	Required	Required	Required	Pre-deposited	Required	Not Required
3. Cascade Attenuation	None	None	Partial	Yes (pool)	None	Yes (locks)[†]
4. Position Transfer	Supply only	Supply only	No	No	Supply only	Both (inverted)
5. Governance Bounds	None	None	None	Immutable	None	$0.5\times$–$2\times$
6. Param. Transitions	Instant	Instant	Instant	N/A	Instant	Asymptotic
7. Oracle Type	Chainlink	Chainlink	CL+OSM	Chainlink	Uniswap	Log TWAP
8. Capacity Rate-Limit	None	None	None	None	None	$\sqrt{n+2}$ scaling
9. Spam Protection	Gas price	Gas price	Gas price	Gas price	Gas price	Gas + PoW
10. Position Caps	None	Isolation	Debt ceil.	Debt ceil.	Borrow caps	Beta-distributed

[†] Attenuation proportional to lock adoption ϕ; without locks, cascade risk equals traditional protocols.

5. **Acma**: An access-control contract extending OpenZeppelin's `AccessManager` [38] with the protocol-specific role catalogue and the `relate()` selector-to-role binder.

6. **WPosition**: Optional ERC20 wrappers for supply/borrow positions, registered per-token via `Pool.enwrap()`; enable composition with external DeFi protocols that expect plain ERC20 transfer semantics (see Section IIb. 9).

Definition I. 3.1 (Minimum Token Requirements). $|\mathcal{T}| \geq 2 \wedge \text{decimals}(T_i) \geq 6$, ensuring sufficient precision for interest calculations and cross-collateralization.

I. 4 Core Mechanisms

I. 4.1 Locked Positions and Cascade Attenuation

Definition I. 4.1 (Position Lock). For each user u and position type $p \in \{\text{supply}, \text{borrow}\}$, the protocol maintains $\text{balance}(u,p)$, $\text{lock}(u,p)$, and $\text{liquid}(u,p) = \text{balance} - \text{lock}$. Operations accept a term parameter `dt_term`; when nonzero, supplied assets cannot be redeemed and borrowed amounts cannot be settled until the slot epoch elapses. The sentinel `dt_term = type(uint256).max` produces a permanent (irrevocable) lock; any other term writes into one of 16 quarterly ring slots that expire automatically. Transfers proportionally move locked amounts:

$$\text{transfer}_{\text{lock}}(u_1, u_2, v) = \left\lfloor \frac{\text{lock}(u_1) \cdot v}{\text{balance}(u_1)} \right\rfloor \tag{1}$$

Lock Terms. Locks are *time-bound by default*: callers pass a term `dt_term` that pins the principal to one of 16 quarterly ring slots (default slot length `MAX_LOCK_TERM` = 3 months, total horizon $\approx$ 48 months). Slots expire automatically and are reclaimed by `free()`. Passing `dt_term = type(uint256).max` upgrades the lock to *permanent* (irrevocable); only this path matches the cascade-attenuation analysis below. Cascade attenuation by factor $(1-\phi)$ therefore applies only to the *permanent-locked* fraction; timed locks contribute attenuation only until their slot epoch elapses. A user can otherwise exit only via transfer (which proportionally moves the lock) or liquidation. Crucially, only the *principal* is locked: accrued interest on locked supply positions can be redeemed at any time, and accrued interest on locked borrow positions can be settled at any time, too. Still, the design creates a fundamental tension: cascade protection depends on widespread adoption, yet rational adoption requires holding periods exceeding 5 years at typical utilization [5]. The protocol addresses this tension by tying APY incentives to utilization levels.

Lock Bonus/Malus. To encourage adoption, locked suppliers earn additional interest $I_{\text{bonus}} = I \cdot \rho_u \cdot r_{\text{bonus}}$, while locked borrowers pay reduced interest $I_{\text{malus}} = I \cdot \rho_u \cdot r_{\text{malus}}$, where $\rho_u = \text{lock}/\text{balance}$ and $r_{\text{bonus}}, r_{\text{malus}} \in [0, s]$. The code enforces $r_{\text{bonus}} \leq s$ and $r_{\text{malus}} \leq s$ independently, where s is the spread parameter. At full lock adoption ($\bar{\rho} = 1$), the combined constraint $r_{\text{bonus}} + r_{\text{malus}} \leq 2s$ holds at the boundary, with protocol margin approaching zero.

Theorem I. 4.1 (Cascade Attenuation). *In a pool where fraction ϕ of supply positions are locked, the maximum cascade amplification factor is bounded by $(1-\phi)$. Locked positions attenuate sell pressure by preventing immediate redemption of seized collateral; they do not eliminate cascades among unlocked positions.*

Because locked supply tokens cannot be redeemed, liquidation transfers position tokens without underlying assets leaving the vault. Only the unlocked fraction $(1-\phi)$ generates sell pressure. A full proof appears in [5]. Simulation confirms that at a 25% price shock, unlocked positions suffer 85.7% liquidation versus 29.4% when fully locked [6].

Liquidation Precedence. In practice, liquidators prefer unlocked positions for their immediate liquidity, creating *de facto* seniority for locked positions. This ordering is market-driven rather than protocol-enforced.

I. 4.2 Position Transfer Semantics

Supply positions follow standard ERC20 semantics: `transfer` pushes value to the receiver, and the health check is applied to the sender. Borrow positions, by contrast, implement *inverted* semantics that reflect the nature of debt: `transfer(from, amount)` **pulls** debt

from the `from` address, `transferFrom` requires *dual* approval from both sender and receiver, and the health check falls on the receiver.

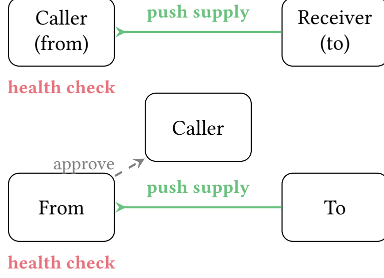


Figure 2: Supply position — standard ERC20 push semantics

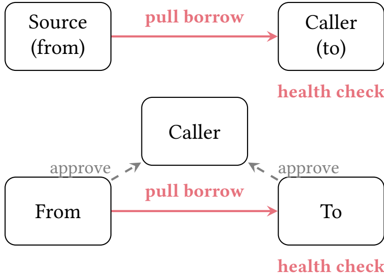


Figure 3: Borrow position — inverted pull semantics with dual approval

Definition I. 4.2 (Debt Transfer)**.** When debt is transferred from user A to user B: (1) A's borrow position decreases by d; (2) B's borrow position increases by d; (3) B must have sufficient collateral; (4) both must approve the intermediary (unless self-transfer).

Rationale. Debt transfer differs fundamentally from asset transfer: with assets, the receiver benefits; with debt, the sender benefits by losing an obligation. The inverted semantics reflect this distinction—standard ERC20 pushes value to the receiver, whereas borrow positions pull debt from the source. The Pool contract as owner bypasses approvals (and health checks) to enable liquidations.

I. 4.3 Lethargic Governance

Definition I. 4.3 (Parameter Transition)**.** A parameter θ transitions from θ_0 to θ_1 subject to $0.5 \cdot \theta_0 \leq \theta_1 \leq 2 \cdot \theta_0$. The effective value follows the time-weighted mean:

$$\bar{\theta}(t) = \theta_1 - \frac{(\theta_1 - \theta_0)(t_s - t_0)}{t - t_0} \tag{2}$$

which asymptotically approaches θ_1 as $t \to \infty$.

Safety Constraints. Each single change is bounded to at most $2\times$, overlapping transitions are disallowed, and changes are rate-limited to once per governance period (monthly). Mandatory initial lock periods ranging from 3 months to 1 year (by category) prevent manipulation at deployment. Full parameter tables are provided in Appendix I. 11.

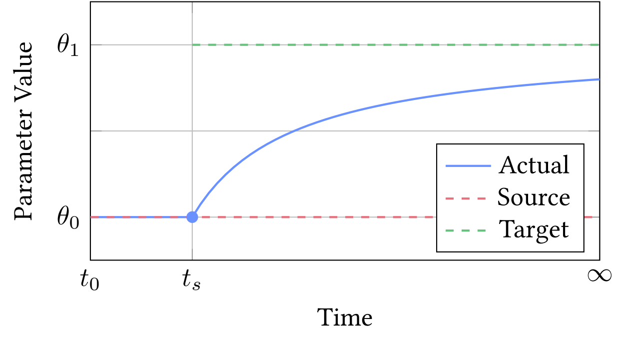


Figure 4: Lethargic parameter transition (asymptotic)

I. 4.4 Beta-Distributed Position Caps

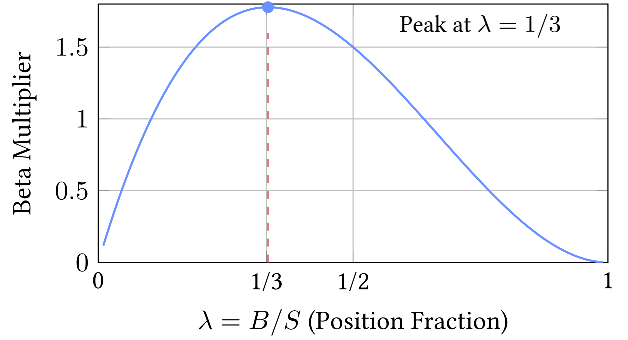


Figure 5: Beta distribution cap function $12\lambda(1-\lambda)^2$

Definition I. 4.4 (Position Cap Function)**.** For balance B, supply S, and n large holders (≥ 1 token unit):

$$\text{cap}(B, S, n) = \frac{C_{\max} \cdot 12\lambda(1-\lambda)^2}{\sqrt{n+2}} \tag{3}$$

where $\lambda = B/S$ and $C_{\max}$ is the maximum *relative* cap.

The offset $+2$ prevents degeneracy at low holder counts. In addition, governance sets a holder floor $n_{\min}$ via the `MIN_HOLDERS` parameter; the effective count is $\max(n, n_{\min})$, which tightens per-user caps during the cold-start phase when few holders have joined. When $\lambda = 0$ (a new user with no existing balance), the beta factor is bypassed and the cap reduces to $C_{\max}/\sqrt{n+2}$, ensuring a non-zero entry allocation.

The Beta(2,3) shape is a design choice, not a uniquely optimal distribution—any unimodal density vanishing at 0 and 1 achieves similar goals. We chose Beta(2,3) for its mode at $\lambda = 1/3$ (favoring medium positions) and analytic tractability. A simpler $4\lambda(1-\lambda)$ (Beta(2,2)) or $\lambda(1-\lambda)$ would also penalize extremes; the asymmetry toward smaller positions is a preference, not a theoretical necessity.

Rate-Limiting, Not Sybil Prevention. The $\sqrt{n+2}$ divisor bounds the *accumulation rate*: k accounts yield $O(\sqrt{k})$ total capacity gain per iteration. This does not, however, penalize long-term equilibrium share—simulation shows that a Sybil attacker retains 48.6% share (versus 50% initial) after 60 weeks, a marginal 1.4% penalty. The mechanism thus prevents rapid capacity monopolization while accepting that patient attackers with sufficient

capital can reach similar long-term positions. Combined with a 7 iterations/week cap, capacity growth remains bounded regardless of account splitting. See [6] for detailed simulations.

I. 4.5 Health Factor and Liquidation

Definition I. 4.5 (Health Factor)**.** For user u with supply values V_s^i and borrow values V_b^i across n tokens:

$$H(u) = \frac{\sum_i w_s^i \cdot V_s^i}{\sum_i w_b^i \cdot V_b^i} \tag{4}$$

where $w_s^i, w_b^i \in [0, 255]$ are uint8 weights. The implementation divides each weighted value by $\texttt{WEIGHT_MAX} = 255$ and averages across tokens; both normalizations cancel in the ratio.

With the default weights $w_s = 170$ and $w_b = 255$, the effective LTV is $170/255 \approx 66.67\%$. This is standard over-collateralization—the same mechanism underlying Compound's and Aave's LTV parameters, expressed through weight ratios rather than an explicit LTV setting. The resulting implicit liquidation bonus $\beta = w_b/w_s - 1 = 50\%$ incentivizes liquidators.

Hybrid Liquidation. The protocol supports both centralized keepers (the default, via `square()`) and governance-enabled public liquidators (PoW-gated, via `liquidate()`). Both paths use a *debt assumption* model in which the liquidator assumes fraction 2^{-e} of the victim's positions. For a given exponent e, both supply tokens $s = \text{supply}(V) \gg e$ and debt $d = \text{borrow}(V) \gg e$ transfer atomically. No liquid capital is required—only sufficient collateral headroom. The liquidator's health is verified post-liquidation.

I. 4.6 Oracle TWAP

The oracle aggregates prices via a log-space exponential moving average (EMA). Each refresh queries the AMM in both directions (source→target and target→source) and computes a bidirectional geomean spread, which eliminates directional bias in asymmetric pools:

$$\overline{m}_t = \alpha \cdot \overline{m}_{t-1} + (1-\alpha) \cdot \log_2(\bar{p}_{t-1}) \tag{5}$$

$$\overline{r}_t = \alpha \cdot \overline{r}_{t-1} + (1-\alpha) \cdot r_{t-1} \tag{6}$$

where $r_t = \frac{1}{2}(\log_2(1 + s_t^{AB}) + \log_2(1 + s_t^{BA}))$. Smoothing in $\log_2$ space produces geometric mean temporal averaging (as in Uniswap V3 [22]). The implementation defers the most recent tick by one period (`twap_.last` → `twap_.mean` on the next refresh) to provide two-tick flash-loan immunity; see `library/TWAP.sol::update`. With $\alpha = 0.944$ (12-period half-life) and hourly refreshes, approximately 40 hours of sustained manipulation achieves 90% price deviation.

Spread Scaling. Large positions incur wider spreads via $\mu = \log_2(2x+2)$ where $x = n \cdot s$, reflecting market impact without requiring per-pair parameters. The base spread itself serves as sensitivity control: liquid pairs scale slowly, while illiquid pairs scale quickly.

Bad-Debt Risk from Stale Pricing. The slow convergence is double-edged. After a genuine 2× price move, only 22.5% is absorbed after 14 hours. Combined with the 1-hour refresh limit, the protocol can be 2+ hours blind during a crash. If collateral drops 50%, the oracle reports near pre-crash prices for hours, potentially delaying liquidations and allowing bad debt to accumulate. The 50% over-collateralization buffer provides margin: at the default 66.67% LTV the protocol sits on the boundary where bad debt begins to emerge under tail-event price crashes (>50%); Monte Carlo simulation and analytical bounds quantify the resulting risk profile. The conservative governance floor of 33% LTV retains zero bounded bad debt for crashes up to 50%. See [6] for TWAP simulations and quantitative bad-debt risk bounds.

I. 4.7 Interest Rates

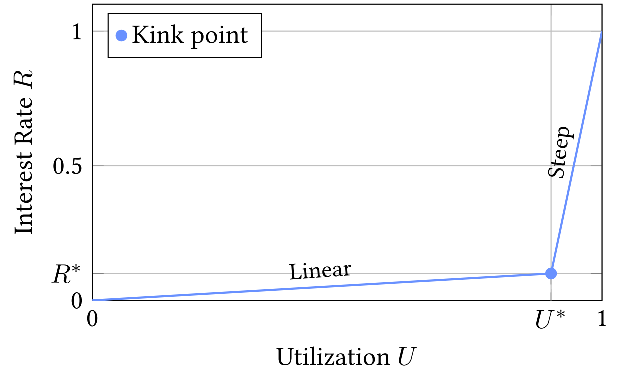


Figure 6: Piecewise-linear interest rate curve

Definition I. 4.6 (Interest Rate Model)**.** For utilization U, optimal utilization U^*, and optimal rate R^*:

$$R(U) = \min(R_{\text{kink}}(U),\ 2)\,,$$
$$R_{\text{kink}}(U) = \begin{cases} UR^*/U^* & \text{if } U \leq U^* \\ \dfrac{U(1-R^*) - (U^* - R^*)}{1-U^*} & \text{otherwise} \end{cases} \tag{7}$$

The implementation caps the base rate at 200% to bound interest in the post-kink regime.

The protocol maintains a symmetric spread: $R_{\text{borrow}} = R_{\text{base}} \cdot (1+s)$, $R_{\text{supply}} = R_{\text{base}} \cdot (1-s)$, generating $2s$ margin on interest flows. See [5] for mathematical foundations including interest accrual and time-weighted integration.

I. 5 Anti-Spam Protection

When governance enables public operations—such as liquidation via `liquidate()` or oracle refresh via `refresh()`—proof-of-work gates prevent spam. The PoW requires:

$$\text{zeros}\big(\text{keccak256}(\text{blockHash} \| \text{tx.origin} \| \text{msg.data})\big) \geq d \tag{8}$$

where each difficulty level requires 16× more computational work. Combined with token-bucket rate limiting (capacity C, regeneration 1/second), this prevents mempool flooding and block stuffing. By default, operations are restricted to authorized keepers; PoW only applies in governance-enabled public mode.

Fairness Limitations. PoW shifts the advantage from capital (gas wars) to computation. Professional miners achieve 10–20× higher hash rates than browser-based implementations, and on PoS chains (e.g., Avalanche) validators who know they will propose a block can pre-compute favorable nonces, creating a validator-advantage vector. In short, PoW provides anti-spam friction rather than fair access—it changes the extraction mechanism from capital to computation without eliminating it.

I. 6 Governance and Parameters

All protocol parameters are governed via the lethargic transition mechanism (Definition I. 4.3). They are organized into four contract domains: Pool (rate limits, PoW difficulty, health weights), Position (interest rate model, caps), Oracle (TWAP behavior, feed management), and Vault (entry/exit fees). Full parameter tables and role categories appear in Appendix I. 11.

Theorem I. 6.1 (Governance Rate Bound). *Under multiplicative bound $\mu = 2$ and minimum cycle $\Delta t_{\min}$, the parameter at time t satisfies:*

$$\mu^{-n}\theta_0 \leq \theta(t) \leq \mu^n\theta_0 \tag{9}$$

where $n = \lfloor (t - t_0)/\Delta t_{\min} \rfloor$. A 10× change requires $\lceil \log_2 10 \rceil = 4$ months.

The protocol implements a three-tier role hierarchy per governable function: an executor (who calls the function), an admin (who grants and revokes the executor role), and a guard (who provides veto-only revocation). Guard roles serve as a "break glass" mechanism, allowing emergency halts of malicious actions without granting execution power. The governance rate bound proof is provided in [5].

I. 7 Security Analysis

I. 7.1 Adversary Model

We consider an adversary $\mathcal{A}$ who controls capital K, hash rate H, arbitrarily many accounts, governance access with probability p_g, and same-block front-running capability. We assume that $\mathcal{A}$ cannot break cryptographic primitives or control more than 50% of consensus.

I. 7.2 Core Security Properties

Cascade Attenuation (Theorem I. 4.1). Because locked supply positions cannot be redeemed, cascade sell pressure is bounded to $(1-\phi)$ of the unlocked pool. This attenuates—but does not prevent—cascades. Simulation confirms that at a 25% price shock, unlocked positions suffer 85.7% liquidation versus 29.4% when fully locked (Section I. 8).

Rate-Limited Accumulation. The $\sqrt{n+2}$ divisor bounds per-iteration capacity gain sublinearly. Combined with 7 iterations/week, reaching 99% capacity requires approximately $850\sqrt{n/100}$ iterations. This constitutes rate-limiting rather than Sybil prevention: the long-term share distribution is ultimately determined by initial capital [6].

Governance Rate Bound (Theorem I. 6.1). Catastrophic parameter changes require 6+ months of sustained malicious governance, providing ample time for detection and response.

Solvency. The code enforces $r_{\text{bonus}} \leq s$ and $r_{\text{malus}} \leq s$ independently per parameter. At full lock adoption, the combined constraint $r_{\text{bonus}} + r_{\text{malus}} \leq 2s$ holds at the boundary, with margin approaching zero. The aggregate solvency formula ($r_{\text{bonus}} \cdot \bar{\rho}^S + r_{\text{malus}} \cdot \bar{\rho}^B \leq 2s$) is a sufficient condition that is not directly enforced on-chain; the per-parameter bounds serve as the operative constraints.

I. 7.3 Risk Assessment

Table 2: Risk severity assessment

Risk Category	Likelihood	Impact
Oracle Manipulation	Medium	High
Governance Attack	Low	High
Liquidation Cascade†	Medium	High
Smart Contract Bug	Medium	Critical
Keeper Centralization	Medium	High
Bad Debt (Extreme Vol.)	Low	Critical
Lock Mechanism Abuse	Low	Medium
PoW Resource Advantage	Medium	Low
Oracle Staleness	Medium	Medium

†Impact conditional on lock adoption; without locks, impact is High.

Oracle. The log-space TWAP with $\alpha = 0.944$ requires approximately 40 hours of sustained manipulation to achieve 90% deviation, and flash loans are ineffective due to the two-tick immunity window. However, the 2+ hour blindness during genuine crashes creates bad-debt risk (Section I. 4.6).

Governance. Parameter changes are bounded by 0.5×–2× per cycle, and guard roles enable emergency revocation. Nevertheless, an attacker who sustains control for 6+ months can achieve significant cumulative drift.

Smart Contracts. The implementation uses reentrancy protection via `ReentrancyGuardTransient`, Solidity 0.8+ overflow checks, and OpenZeppelin access control. No formal verification has been performed (Section IIb. 9).

Full proofs of the above properties are provided in [5].

I. 8 Evaluation

I. 8.1 Agent-Based Cascade Simulation

We simulate 1,000 borrowers with log-normal positions, health factors $H \sim \mathcal{N}(1.5, 0.3)$ truncated at $[1.0, 3.0]$, and

linear price impact $\Delta p = k \cdot V_{\text{sold}}$ with $k = 1.5 \times 10^{-4}$ (pool ~25% of market depth).

Table 3: Cascade simulation — positions liquidated (%), $k = 1.5 \times 10^{-4}$

Lock ϕ	10%	15%	20%	25%	30%
0%	8.6	19.8	36.5	85.7	99.1
25%	8.2	16.5	30.6	55.8	84.9
50%	7.8	14.4	26.7	37.1	61.2
75%	7.8	13.5	22.4	33.3	47.9
100%	7.4	12.7	19.8	29.4	38.6

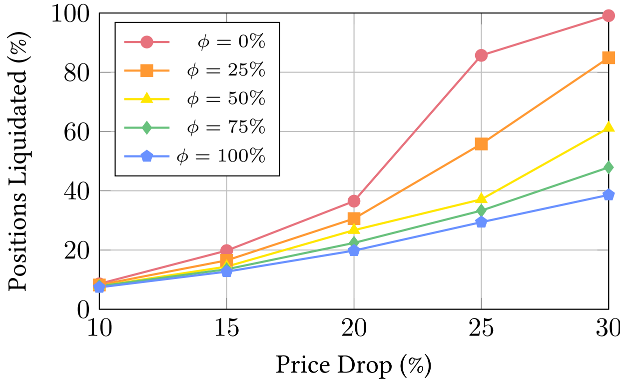


Figure 7: Cascade depth vs. price drop for varying lock fractions

Simulation Limitations. The simulation uses 1,000 agents, which is too few for high statistical confidence. Health factor distributions are assumed rather than calibrated against real DeFi data. The linear price impact model [47] is a first-order approximation; real order books exhibit concave impact. No confidence intervals or sensitivity analysis on k are provided, and historical stress events are discussed narratively but not backtested. Implementation details appear in [6].

I. 8.2 Gas Cost Analysis

Gas measurements are post-optimization Foundry snapshots (`FOUNDRY_OPTIMIZER=true`, default profile, `via_ir=false`), benchmarked against Aave V3.3 figures from a 2025 Cyfrin gas-optimization audit. Banq's warm-path gas is competitive with mainstream lending protocols on supply and dominates them on every other operation:

Table 4: Banq vs. Aave V3 warm-path gas (Foundry, optimizer ON)

Operation	Banq	Aave V3	Δ
Supply	167,209	146,354	+14%
Borrow	140,884	247,485	−43%
Repay (settle)	123,403	189,518	−35%
Withdraw (redeem)	128,380	181,430	−29%
Liquidation (full)	298,882	389,059	−23%

The savings come from two implementation choices: (i) the log-space compounding index ([4]) eliminates `exp()` from the write path, saving ~1,200 gas per accrual at the cost of ~1,100 gas per `totalOf` read; (ii) the packed `_state` (global) and `_stateOf` (per-user) storage words pack `[uint80 index | uint64 stamp/depth | uint112 balance]` into a single SSTORE, halving the write cost relative to the canonical three-slot layout. The `Lock` library applies the same technique: `uint128[16]` ring slots pack two epoch-value pairs per word (8 words instead of 16 per user), and a single `cache` word packs `[uint120 perma | uint120 total | uint16 bits]`. On Avalanche (approximately 1.55 gwei, AVAX <$10), all operations cost well under 1 cent USD.

Table 5: Cold-vs-warm and Banq-unique operations

Operation	Cold	Warm	Notes
Supply	285,814	167,209	—
Borrow	244,833	140,884	self-pair
Borrow (cross-pair)	—	194,317	+53k oracle leg
Liquidation (full)	298,882	—	—
healthOf (view)	44,611	—	view-only
lockSupply (timed)	179,860	66,688	ring-slot lock
lockSupply (perma)	86,712	—	irrevocable
lockBorrow (timed)	180,904	67,720	debt lock
xfer_supply (1 slot)	191,463	—	lock-aware ERC20
xfer_supply (16 slots)	402,525	—	worst-case ring scan

I. 9 Limitations and Future Work

Capital Efficiency. The default 66.67% LTV requires $1.50 of collateral per $1 borrowed—comparable to Compound's 75% and within 25 percentage points of Aave V3 E-Mode (93%). Lethargic governance (§I. 4.3) bounds each cycle to $0.5\times$–$2\times$ the prior target, so the default is reachable down to a 33% conservative floor in one cycle, and the bad-debt analysis in [6] characterises the risk profile across the LTV configurations swept therein; no analysis is provided of what LTV range the target market (XPower tokens on Avalanche) actually requires.

Formal Verification. The smart contracts have not undergone formal verification. Given the 10 interacting mechanisms—where locks affect caps, which affect health, which affects liquidation—this represents a significant gap that undermines the security analysis. Formal proofs of health factor correctness, position token conservation, and interest accrual monotonicity are absent. This is the highest-priority future work, recommended via Certora, Halmos, or the K framework.

Composability Risks. Wrapped positions (WPosition) interact with external DeFi protocols, yet no analysis addresses how inverted borrow semantics or irrevocable locks behave under composition. External contracts that expect standard ERC20 semantics may mishandle borrow

positions.

Missing Game-Theoretic Analysis. No model addresses competitive liquidator dynamics under PoW constraints, equilibrium liquidation profit, or oracle sandwich attack quantification. While PoW raises the cost of sandwich attacks, it does not eliminate them.

Future Directions. Areas for further development include dynamic parameter adjustment based on market conditions, cross-chain lending via message passing, privacy-preserving positions via zero-knowledge proofs, and integration of additional oracle sources for off-chain price anchoring.

I. 10 Conclusion

XPower Banq presents a DeFi lending protocol that balances capital efficiency against bounded bad-debt risk. Its strongest contributions are lethargic governance—a genuine innovation with formal rate bounds—and the log-space geometric-mean TWAP oracle, which is both novel and empirically validated. The locked position mechanism provides cascade attenuation proportional to adoption, though its effectiveness depends on rational adoption dynamics that remain uncertain [5]. The beta-distributed cap function rate-limits capacity accumulation but does not prevent long-term Sybil advantage. The default 66.67% LTV positions the protocol within range of mainstream lending markets while remaining adjustable via lethargic governance (§I. 4.3). Formal verification and game-theoretic liquidation modeling remain the most significant gaps for future work.

I. 11 Protocol Parameters

I. 11.1 Pool Parameters

Table 6: Pool parameters. Time units: s=second, d=day, w=week, m=month, y=year.

Parameter	Setting	Description
MAX_SUPPLY	1w ∈ [1s, 1y]	Max supply rate-limit
MIN_SUPPLY	1d ∈ [1s, 1y]	Min supply rate-limit
POW_SUPPLY	0 ∈ [0, 64]	Supply PoW difficulty
MAX_BORROW	1w ∈ [1s, 1y]	Max borrow rate-limit
MIN_BORROW	1d ∈ [1s, 1y]	Min borrow rate-limit
POW_BORROW	0 ∈ [0, 64]	Borrow PoW difficulty
POW_SQUARE	0 ∈ [0, 64]	Liquid. PoW difficulty
WEIGHT_SUPPLY	170 ∈ [0, 255]	Supply health weight‡
WEIGHT_BORROW	255 ∈ [0, 255]	Borrow health weight

‡Deployer-set; reference deployments use 170/255 ≈ 66.67% LTV.

I. 11.2 Position Parameters

Table 7: Position parameters

Parameter	Setting	Description
UTIL	90% ∈ [0, 100%]	Optimal utilization
RATE	10% ∈ [0, 100%]	Rate at kink
SPREAD	10% ∈ [0, 50%]	Rate half-spread§
LOCK_BONUS	10% ∈ [0, SPREAD]	Locked supply APY bonus
LOCK_MALUS	10% ∈ [0, SPREAD]	Locked borrow APY reduction
CAP	– ∈ $[0, 2^{112}-1]$	Position cap (≈ 5.19×10^{15} at 18 decimals)
MIN_HOLDERS	– ∈ $[0, 10^{18}]$	Min large holders

§Lower-bounded by LOCK_BONUS and LOCK_MALUS.

I. 11.3 Oracle Parameters

Table 8: Oracle parameters

Parameter	Setting	Description
DECAY	0.944 ∈ [0.5, 1.0]	EMA decay
LIMIT	1h ∈ [1s, 1d]	Min refresh interval
LEVEL	0 ∈ [0, 64]	Refresh PoW difficulty
DELAY	14d ∈ [1w, 3m]	Feed enlist timelock

I. 11.4 Vault Parameters

Table 9: Vault parameters

Parameter	Setting	Description
FEE_ENTRY	0.1% ∈ [0, 50%]	Deposit fee
FEE_EXIT	1.0% ∈ [0, 50%]	Withdrawal fee

I. 11.5 Change Rate Constraints

Table 10: Initial lock periods of parameters

Category	Lock Period
Interest rate model	3m
Oracle TWAP settings	3m
Oracle feed delay	1y
Vault fees	3m
Pool weights	3m
Pool rate limits	1y

Table 11: Maximum parameter change over time

Cycles	Max Increase	Max Decrease
1	2×	0.5×
2	4×	0.25×
3	8×	0.125×
6	64×	0.016×
12	4096×	0.00024×

I. 11.6 Role-Based Access Control

The protocol implements a three-tier role hierarchy per governable function: executor (calls the restricted function), admin (grants/revokes the executor role), and guard (veto-only revocation).

Governance Attack Precautions. Delayed feed changes require up to 3 months. Guard role separation prevents escalation. Gradual parameter migration eliminates arbitrage from sudden changes. Bounded cumulative change creates predictable worst-case scenarios. Immutable constraints include minimum decimals ≥ 6, minimum 2 tokens, a fixed token list, and non-upgradeable contracts.

I. 12 Protocol Constants

The following tables list all compile-time constants defined in the protocol's `Constant.sol` library.

Table 12: Time unit constants (in seconds)

Constant	Value	Definition
CENTURY	3,155,760,000	$365.25 \times 100 \times 86400$
YEAR	31,557,600	365.25×86400
MONTH	2,629,800	YEAR / 12
WEEK	604,800	7×86400
DAY	86,400	24×3600
HOUR	3,600	60×60
MINUTE	60	60 seconds
SECOND	1	1 second

Table 13: Fixed-point representation constants

Constant	Value	Meaning
ONE	10^{18}	WAD unit (1.0)
TWO	2×10^{18}	WAD unit (2.0)
HLF	0.5×10^{18}	WAD unit (0.5)
NIL	0	WAD unit (0.0)
PCT	10^{16}	1 percentage point
BPS	10^{14}	1 basis point

Table 14: Protocol constraints and limits

Constant	Value	Purpose
MAX_DIFFICULTY	64	Max PoW difficulty
MIN_HOLDERS	10^{18}	Min holders upper bound
VERSION	0x1ba	Protocol version

I. 13 EMA Decay Factors

Table 15 lists pre-computed EMA decay factors relating half-life (in refresh periods) to the decay coefficient α, used for log-space TWAP oracle smoothing.

Table 15: Pre-computed EMA decay factors

Half-life	Decay α	Use Case
1	0.500000	Fast response
2	0.707107	Short-term
12	0.943874	Medium-term
24	0.971532	Long-term

XPower Banq: Part IIa

Ring-Buffer Time Locks

A 16-Slot Quarterly Ring Buffer with $O(1)$ Depth Tracking for Graduated Commitment in DeFi Lending

Abstract

DeFi protocols benefit from knowing not just *whether* tokens are locked but *how long* they are committed. Naive implementations require $O(n)$ storage walks or unbounded gas. We present a 16-slot quarterly ring buffer that stores time-locked positions in constant per-slot storage with bitmap-guided $O(k)$ sweep ($k \leq 16$). The key contribution is an $O(1)$ cached depth metric: by storing the epoch-weighted sum $\Sigma = \sum v_i(e_i+1)$, the exact token-second integral $D = \Sigma Q - Tt + pL$ can be reconstructed in two SLOADs. The implementation returns the normalized form $\widehat{D} = \lfloor \Sigma Q/L \rfloor - \lfloor Tt/L \rfloor + p$ (token-equivalent units, suitable for direct comparison with `balanceOf`) using two `Math.mulDiv` calls and no iteration. Each lock requires 1–3 SSTOREs; queries are $O(1)$ or $O(k)$; proportional transfer runs in $O(k)$. Storage is aggressively packed: per-user state occupies 10 storage words (8 ring + cache + depth), with the `cache` word holding three logical fields (`perma`, `total`, `bits`) in a single SLOAD. The mechanism is deployed in the XPower Banq lending protocol [1], where the lock depth drives graduated lock bonus/malus on interest rates. Measured gas: 66,688 warm / 179,860 cold for a timed `lockSupply`, 86,712 cold for a permanent lock.

Keywords: time-locked positions, ring buffer, token-seconds, DeFi lending, graduated commitment, Solidity

IIa. 1 Introduction

Time-locked positions are a recurring primitive in DeFi. Vote-escrow protocols (Curve veCRV [34]), liquid staking wrappers (Convex vlCVX [35]), and lending protocols all require users to commit tokens for a bounded duration. Beyond binary locked/unlocked status, protocols benefit from a measure of *commitment depth*—the integral of locked amount over remaining time, which we call *token-seconds*.

Computing token-seconds on-chain presents a tension: the naive $O(n)$ summation over all lock entries is unbounded in gas, while maintaining a single aggregated counter sacrifices the ability to support multiple concurrent locks with different expiry dates.

Contributions. We resolve this tension with two mechanisms layered on a shared ring-buffer data structure:

1. **Ring-Lock**: A 16-slot quarterly ring buffer with a bitmap-guided $O(k)$ sweep for expired slots, providing bounded-gas lock creation and expiry (Section IIa. 4).
2. **Time-Lock**: An extension that adds a single stored value—the epoch-weighted sum $\Sigma = \sum v_i(e_i{+}1)$—enabling $O(1)$ reconstruction of exact token-seconds via an algebraic identity (Section IIa. 5).

The protocol context is the XPower Banq lending protocol [1], where lock depth drives graduated interest rate adjustments (Section IIa. 8). The locked-positions mechanism is described in [1] §4.1 and the interest rate application in [1] §4.7. Formal proofs of the ring-buffer invariants and depth identity appear in Section IIa. 6. A companion document [5, 6, 7] provides extended mathematical analysis.

IIa. 2 Related Work

Curve veCRV [34]. Fixed 4-year linear decay from a single lock slot. $O(1)$ query via closed-form decay, but no support for multiple concurrent locks or variable duration. The bias/slope model requires periodic global checkpoints.

Convex/Aura vlCVX [35]. Epoch-based unlock queues with linear scan for expired epochs. Supports multiple lock durations, but sweep cost is $O(n)$ in the number of distinct epochs—unbounded without a cap on lock granularity.

Uniswap v3 NFT Locks. Per-position NFTs carry individual timestamps. No aggregation across positions; each lock is an independent storage entity.

OpenZeppelin TimelockController [36]. Designed for governance action delays, not position locks. Single-use: one pending action per hash, no ring or aggregation structure.

Ring Buffers. Circular buffers are classical in systems programming [40]: OS scheduling, network packet buffers, Uniswap V3 oracle observation arrays. Our application to on-chain time locks with bitmap-guided sweep and cached depth appears to be novel.

Table 16: Comparison of time-lock mechanisms.

	Duration Range	Storage per User	Lock Cost	Query Cost	Depth Metric
veCRV	Fixed 4y	1 word	$O(1)$	$O(1)$	Linear
vlCVX	Epochs	$O(n)$	$O(1)$	$O(n)$	None
NFT	Any	$O(n)$	$O(1)$	$O(n)$	None
Ours	$[0, 16Q)$	10 words	$O(1)$	$O(1)$	Token·s

IIa. 3 Preliminaries

IIa. 3.1 Definitions

Definition IIa. 3.1 (Epoch)**.** The *absolute epoch index* at timestamp t is $e = \lfloor t/Q \rfloor$, where Q = LOCK_TERM $\approx$ 91.3 days (one calendar quarter).

Definition IIa. 3.2 (Slot Index)**.** The ring-buffer *slot index* for epoch e is $i = e \bmod 16$.

Definition IIa. 3.3 (Lock Expiry)**.** A lock in epoch e expires at timestamp $(e{+}1) \times Q$, the start of the next epoch.

Definition IIa. 3.4 (Token-Seconds)**.** The *token-second depth* for user u at timestamp t is:

$$D(u,t) = \sum_{i \in A} v_i\big((e_i{+}1)Q - t\big) + p \cdot L \qquad (10)$$

where A is the set of active (non-expired) timed slots, v_i and e_i are the value and epoch of slot i, $p = \texttt{perma}[u]$ is the permanently locked amount, and $L = \text{LOCK_TIME} = 16Q \approx 48$ months.

IIa. 3.2 Storage Layout

The complete storage is 10 words per user (most zero-initialized):

```
1 // 128-bit packed words: [uint16 epoch|uint112 value].
2 // uint128[16] = 8 storage words, two slots per word.
3 struct Lock {
4   // 16-slot quarterly ring buffer (8 words, 2 per word)
5   mapping(address => uint128[16]) slots;
6   // [uint120 perma | uint120 total | uint16 bits]
7   mapping(address => uint256)     cache;
8   // epoch-weighted sum: sigma v_i*(e_i+1)
9   mapping(address => uint256)     depth;
10 }
```

Listing 1: Lock storage layout

The first 9 words (8 ring slots + `cache`) constitute the Ring-Lock layer. The `depth` mapping is the Time-Lock extension. The `cache` word merges the irrevocable `perma` amount, the cached ring `total`, and the active-slot `bits` bitmap into a single SLOAD/SSTORE—a key gas optimization, since most operations touch all three fields together. Per-call amount is bounded by `uint112`; `perma` saturates at $2^{120}-1$ (cumulative permanent deposits would need ~256 max-sized adds before saturation).

IIa. 3.3 Bitmap Encoding

The `cache` word packs three values: the upper 120 bits store the irrevocable `perma` amount, the middle 120 bits

store the cached ring `total`, and the lower 16 bits store the active-slot bitmap. Decoding:

$$\texttt{perma} = \texttt{cache} \gg 136 \tag{11}$$

$$\texttt{total} = (\texttt{cache} \gg 16) \,\&\, (2^{120}-1) \tag{12}$$

$$\texttt{bits} = \texttt{cache} \,\&\, \texttt{0xFFFF} \tag{13}$$

LSB extraction uses a de Bruijn sequence [39]: the lowest set bit is isolated via $\texttt{lsb} = b \,\&\, (\neg b + 1)$, then multiplied by the constant `0x09AF` modulo 2^{16}, producing a unique 4-bit hash in the top nibble that indexes into a 64-bit lookup table. This replaces 16 conditional SLOADs with 1 SLOAD plus bit arithmetic.

IIa. 4 Ring-Lock Mechanism

This section describes the base ring-buffer operations without depth tracking. Each subsection presents the algorithm, a TikZ figure of the ring-buffer state change, and a complexity analysis.

IIa. 4.1 `more(user, amount, dt_term)` — $O(1)$ Lock Addition

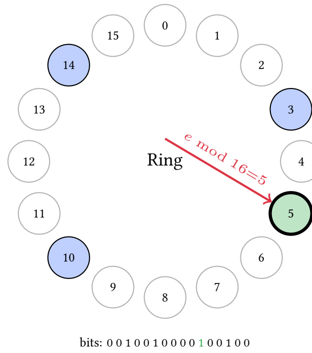


Figure 8: `more()` — the target epoch maps to slot 5; existing active slots (3, 10, 14) are unaffected.

The idea is straightforward: given a lock duration, compute which future epoch the tokens will expire in, then map that epoch to one of the 16 ring-buffer slots via modular arithmetic. If the slot already holds tokens for the same epoch, the new amount is simply added; if it is stale (left over from a previous cycle), the old residue is cleaned up first—this "self-healing" step keeps the cached total consistent without requiring a separate garbage-collection pass.

Algorithm.

0. Require `amount` $\leq 2^{112}-1$ and `dt_term` > 0. Revert otherwise.
1. If `dt_term` $= 2^{256}-1$ (sentinel): execute **permanent path**—add to the `perma` field of `cache` (single SSTORE, since `perma`, `total`, and `bits` share one word); no slot write, no bitmap change. Return. *(Early exit avoids overflow in the addition $t +$ `dt_term` below.)*
2. Require `dt_term` $\leq L - Q$. Revert otherwise.
3. Compute target epoch: $e = \lfloor (t + \texttt{dt_term})/Q \rfloor$.
4. Compute slot index: $i = e \bmod 16$.
5. If `slot.epoch` $\neq e$ (stale or empty):
   - If the old value is positive, subtract it from `total` (self-healing).
   - Overwrite: `slot.epoch` $\leftarrow e$, `slot.value` $\leftarrow$ `amount`.
6. If `slot.epoch` $= e$ (accumulate): `slot.value += amount`.
7. Update `cache`: add `amount` to the `total` field, set bit i in `bits`.

Complexity: 2 SSTOREs (slot + `cache`). Permanent: 1 SSTORE (`cache` only; `perma` + `total` share the word; cold-write gas dominates the first deposit). Measured: 179,860 / 66,688 gas (cold/warm) for timed `lockSupply`, 86,712 cold for permanent.

IIa. 4.2 `free(user)` — $O(k)$ Expired Slot Sweep

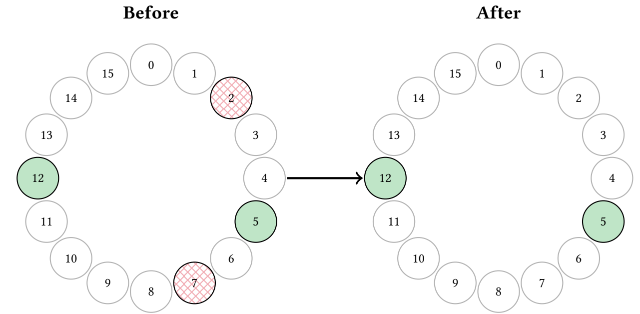


Figure 9: `free()` — bitmap-guided sweep deletes expired slots (2, 7) and clears their bitmap bits, leaving active slots (5, 12) intact.

Because `more()` never deletes old slots, expired tokens can linger in the ring buffer. `free()` walks only the slots marked by the bitmap—skipping empty positions entirely—and removes any whose epoch has already passed. The bitmap serves as a compact "to-do list," so the sweep touches exactly the slots that matter and nothing else.

Algorithm.

1. Read `cache` $\rightarrow$ extract `bits`. (`perma` and `total` are re-loaded just before the cache write-back.)
2. Iterate bitmap via LSB extraction: for each set bit i, load `slots[user][i]`.
3. If `slot.epoch` $< e_{\text{now}}$: accumulate freed amount, delete slot, clear bit i.
4. Write back `cache` with updated `total` and `bits` (`perma` preserved).

Complexity: k SLOADs + up to k SSTOREs (slots; only deleted/decremented slots) + up to 1 SSTORE (`cache`; skipped if neither `total` nor `bits` changed), where k is the number of set bitmap bits. Measured worst-case (`free` 16 expired slots): 31,157 gas.

IIa. 4.3 `push(src, tgt, mul, div)` — $O(k)$ Proportional Transfer

Transferring locks between two users must preserve both the amounts *and* their expiry schedule. `push()` achieves

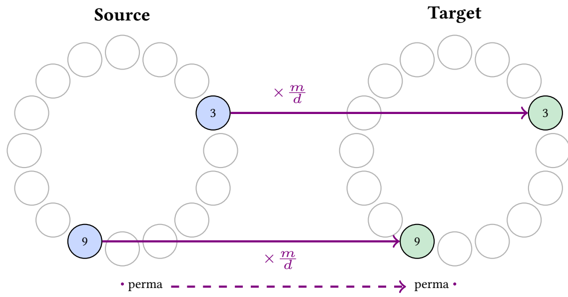


Figure 10: `push()` — proportional transfer of locks from source to target. Each active slot is fractionally moved; permanent locks transfer separately.

this by walking the source's bitmap and moving a proportional fraction m/d of each slot's value into the corresponding slot of the target—same epoch, same remaining duration. The fraction is rounded down per slot to keep all values integral.

Algorithm.

1. Walk source bitmap: for each slot, compute $\Delta v = \lfloor v \cdot m/d \rfloor$.
2. Set target slot: if same epoch → accumulate, else overwrite.
3. Set source slot: subtract Δv; clear bit if zero.
4. Batch-write both `cache` words (`perma`, `total`, `bits` together per side).

Precondition: Caller **must** call `free(tgt)` before `push()` to avoid stale-slot corruption (see Theorem IIa. 6.9).

Complexity: $(2k{+}2)$ SLOADs ($2k$ slot reads + 2 `cache` reads, one per side) + $2k$ SSTOREs + 2 SSTOREs (`cache`×2; `perma` rides along in the same word). Measured `xfer_supply`: 145,251 (perma only) / 191,463 (1 slot) / 402,525 (16 slots, worst case).

IIa. 4.4 `stateAt(user, stamp)` — $O(k)$ Exact Query

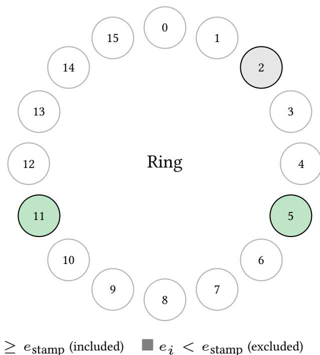


Figure 11: `stateAt()` — slots with epoch $\geq e_{\text{stamp}}$ are included in the total; earlier slots are excluded.

While `totalOf()` returns a cheap cached answer, `stateAt()` computes the *exact* locked balance at an arbitrary timestamp by re-scanning the ring buffer. It includes only those slots whose epoch is at or after the query point, thereby filtering out any tokens that will have expired by that time.

Algorithm.

1. Read `cache` once; start with `total` = `cache.perma`.
2. Walk bitmap: for each active slot with $e_i \geq e_{\text{stamp}}$, add v_i to total.
3. Return total.

Complexity: 1 SLOAD (`cache`, supplies `perma` and `bits`; `total` is recomputed from active slots) + k SLOADs (bitmap-guided slot reads). No writes. Measured `lockStateAt` (16 slots view): 17,895 gas.

IIa. 4.5 `roll(user, amount, dt_term)` — $O(k)$ Sweep-and-Re-Lock

`roll()` combines a partial sweep with a new lock in a single call: it drains up to `amount` tokens from the user's earlier active slots (those whose epoch is strictly before the target epoch) and re-locks the drained amount into the target slot. This is useful for extending an existing position's commitment without making separate `free()` and `more()` calls.

Algorithm.

1. Compute target stamp `stamp` $= t{+}$`dt_term` and target epoch $e_{\text{tgt}} = \lfloor \texttt{stamp}/Q \rfloor$.
2. Walk the user's bitmap and, for each active slot with $e_i < e_{\text{tgt}}$, drain up to the remaining `amount` (using the same `_freeAt` helper as `free()`).
3. Re-lock the drained total into the target slot via `_more` (same code path as `more()`, including self-healing).

Complexity: $O(k)$ — one `_freeAt` pass plus one `_more` call. Same `dt_term` semantics as `more()`.

IIa. 4.6 `totalOf(user)` — $O(1)$ Cached Query

Decode the middle 120 bits of `cache`: a single SLOAD.

Remark IIa. 4.1. The cached total may include expired-but-unswept slot values. Thus `totalOf`$(u) \geq$ `stateAt`(u, t_{now})`.total`. Equality is restored after `free`(u).

IIa. 5 Time-Lock Extension

The Time-Lock layer adds the `depth` mapping to the Ring-Lock structure. Each subsection describes the delta from the corresponding Ring-Lock operation.

IIa. 5.1 The Token-Seconds Integral

Token-seconds measures total commitment depth:

$$D(u,t) = \sum_{i \in A} v_i\big((e_i{+}1)Q - t\big) + p \cdot L \qquad (14)$$

where A is the set of active timed slots, $p =$ `perma`$[u]$, and $L = 16Q$.

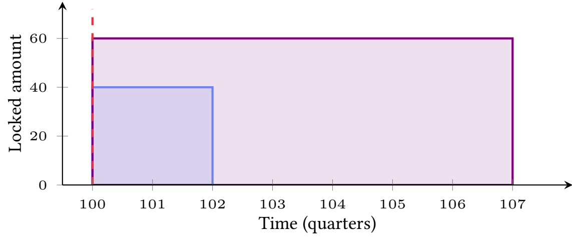


Figure 12: Token-seconds as area under the lock curve. Two timed locks: 40 tokens for 2 quarters (blue) and 60 tokens for 7 quarters (purple). Shaded area = depth.

IIa. 5.2 The $O(1)$ Depth Identity

Theorem IIa. 5.1 (Depth Identity). *Let A be the set of active timed slots for user u, and define:*

$$\Sigma = \textstyle\sum_{i\in A} v_i(e_i{+}1) \quad \textit{(stored in depth)} \tag{15}$$

$$T = \textstyle\sum_{i\in A} v_i \quad \textit{(timed = total − perma)} \tag{16}$$

Then $D(u,t) = \Sigma \cdot Q - T \cdot t + p \cdot L$.

Proof. Expand the sum in (14):

$$\begin{aligned} D &= \sum_{i\in A} v_i\big((e_i{+}1)Q - t\big) + pL \\ &= Q\sum_{i\in A} v_i(e_i{+}1) - t\sum_{i\in A} v_i + pL \\ &= \Sigma Q - Tt + pL \end{aligned}$$

□

Implementation form. To match the token-equivalent units used by the bonus/malus layer (§IIa. 8), the implementation returns the normalized quantity $\widehat{D} = D/L + p$ rather than D itself. Both intermediate products are performed with OpenZeppelin's `Math.mulDiv` [37], which carries a 512-bit numerator natively so no separate sub-epoch remainder term is needed:

$$\texttt{gross} = \lfloor \Sigma \cdot Q / L \rfloor \tag{17}$$

$$\texttt{spent} = \lfloor \textit{total} \cdot t / L \rfloor \tag{18}$$

$$\texttt{depthOf}(u) = \texttt{gross} - \texttt{spent} + p \tag{19}$$

where `total` (the timed total stored in `cache`, distinct from `perma`) is exactly T from (16).

Lemma IIa. 5.2 (Overflow-Safe Computation). *Equations (17)–(19) compute $\lfloor \Sigma Q/L \rfloor - \lfloor Tt/L \rfloor + p$ exactly via two independent 512-bit `mulDiv` operations.*

Proof. `Math.mulDiv`(a,b,c) returns $\lfloor a\cdot b/c \rfloor$ for any 256-bit a, b, c without intermediate overflow, because the product $a\cdot b$ is held in a 512-bit pair before the final division [37]. Applied to (Σ, Q, L) and $(\textit{total}, t, L)$, this directly yields the two floors above. The closing addition of the 120-bit `perma` field cannot exceed 2^{256}. □

Lemma IIa. 5.3 (Σ Overflow Safety). *Let S denote the maximum per-user locked total and E the current epoch. If $S \cdot (E{+}16) < 2^{256}$, then the stored Σ fits in `uint256`. The intermediate product $\Sigma \cdot Q$ in (17) is then carried by `Math.mulDiv` with full 512-bit precision and never materializes as a single 256-bit value.*

Proof. All active epochs lie in $[E, E{+}15]$ (Theorem IIa. 6.1), so $(e_i{+}1) \leq E{+}16$. The sum of slot values satisfies $T = \sum v_i \leq S$. Thus $\Sigma = \sum v_i(e_i{+}1) \leq S(E{+}16) < 2^{256}$. The subsequent product $\Sigma \cdot Q$ in `mulDiv` is held in a 512-bit register before the division by L, so no further overflow check is needed.

Practical margin. The slot epoch field is a `uint16` (§IIa. 3.2), so $E < 2^{16}$. With $E{+}16 < 2^{16}$ this leaves $S < 2^{240}$—far above any realistic ERC-20 token supply. The Position layer's `cap()` mechanism provides the enforcement point. □

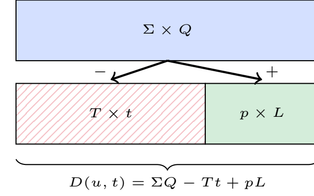


Figure 13: `depthOf()` — geometric decomposition of the $O(1)$ depth reconstruction. The large rectangle ΣQ is reduced by Tt and augmented by pL.

IIa. 5.3 `more()` with Depth Tracking

When a new lock is added, the cached depth accumulator Σ must reflect the change. If the target slot was stale, its old contribution is first subtracted; then the new tokens' contribution—weighted by their expiry epoch—is added. This mirrors the self-healing logic of the base `more()`, extended to one additional storage word.

Delta from §IIa. 4.1: Step 0 (permanent sentinel) returns before reaching the depth update—`depth` is untouched for permanent locks. For timed locks, after the slot write, update `ls.depth[user]`:

- Stale overwrite: $\Sigma \mathrel{-}= v_{\text{old}} \cdot (e_{\text{old}}{+}1)$.
- Then: $\Sigma \mathrel{+}= \text{amount} \cdot (e_{\text{new}}{+}1)$.

Complexity: 3 SSTOREs (slot + cache + depth). Permanent: 1 SSTORE (same as Ring-Lock—`depth` unchanged for permanent locks).

IIa. 5.4 `free()` with Depth Tracking

As expired slots are swept away, their epoch-weighted contributions must also leave the depth accumulator. The sweep piggybacks on the same bitmap walk: each deleted slot's $v_i(e_i{+}1)$ term is collected into a single delta, then subtracted in one storage write at the end.

Delta from §IIa. 4.2: Accumulate $\Delta\Sigma = \sum_{\text{expired}} v_i(e_i{+}1)$ during the sweep. After the loop: `depth`$[u] \mathrel{-}= \Delta\Sigma$.

Complexity: k SSTOREs (slots) + 2 SSTOREs (cache + `depth`).

IIa. 5.5 `push()` with Depth Tracking

Because depth is an additive quantity, transferring locks between users is a zero-sum operation on Σ: whatever

epoch-weighted value leaves the source's accumulator enters the target's. The loop collects the total delta once, then applies symmetric updates to both depth words.

Delta from §IIa. 4.3: Within the source-bitmap loop, accumulate $\Delta\Sigma = \sum \Delta v_i \cdot (e_i{+}1)$. After the loop:

$$\texttt{depth[src]} \mathrel{-}= \Delta\Sigma$$
$$\texttt{depth[tgt]} \mathrel{+}= \Delta\Sigma$$

Complexity: $2k$ SSTOREs + 4 SSTOREs (cache×2, depth×2; perma packed in cache).

IIa. 5.6 `stateAt()` with Depth

The exact query now returns a depth alongside the total. For each included slot the remaining token·seconds contribution $(e_i{+}1)Q - t_{\text{eff}}$ is accumulated on the fly, plus the permanent lock's constant contribution $p \cdot L$.

Delta from §IIa. 4.4: Returns (total, depth). For each active slot with $e_i \geq e_{\text{stamp}}$:

$$\texttt{depth} \mathrel{+}= v_i\big((e_i{+}1)Q - t_{\text{eff}}\big)$$

where $t_{\text{eff}} = \max(t_{\text{stamp}}, t_{\text{now}})$. Permanent contribution: $\texttt{depth} \mathrel{+}= p \cdot L$.

Units note: The accumulator above is in raw token-seconds, but `stateAt` applies the same $/L$ normalization as `depthOf` (§IIa. 5.7, eq. on p. 21) before returning, so the reported `depth` is in token-equivalent units (committed tokens remaining), not token-seconds.

Complexity: k SLOADs + multiply/add per slot. No additional storage reads.

IIa. 5.7 `depthOf(user)` — $O(1)$ Cached Depth

This is the key operation. Rather than looping over every active slot, `depthOf()` reconstructs the token-seconds value from three cached storage words (Σ, total, p) and the current timestamp, using the Depth Identity. To match the units used by the bonus/malus layer (§IIa. 8), the function returns the normalized form

$$\widehat{D}(u,t) \;=\; \lfloor \Sigma \cdot Q/L \rfloor \;-\; \lfloor \mathit{total} \cdot t/L \rfloor \;+\; p$$

i.e. $D/L + p$ expressed in token units (with `mulDiv` performing each division with full 512-bit intermediate precision so no overflow can occur). The result is a single $O(1)$ read with no loops and no bitmap traversal.

Full algorithm:

```
function depthOf(Lock storage ls, address user)
    internal view returns (uint256)
{
    (uint120 perma, uint120 total,) = _full(ls.cache[user]);
    uint256 depth = ls.depth[user];
    uint256 gross = Math.mulDiv(
        depth, LOCK_TERM, LOCK_TIME);
    uint256 spent = Math.mulDiv(
        total, block.timestamp, LOCK_TIME);
    return add256(sub256(gross, spent), perma);
}
```

Listing 2: `depthOf` implementation

Note that `cache` stores `perma` and `total` as *independent* fields (`total` is the timed total; permanent locks are not part of it), so no $\mathit{total} - \mathit{perma}$ subtraction is needed. The single `Math.mulDiv` call replaces the older `mulDiv`/`mulmod` decomposition with a 512-bit native pathway.

Complexity: 2 SLOADs (cache supplies perma + total; depth loaded separately). No loops, no bitmap iteration.

IIa. 5.8 Worked Example

Definition IIa. 5.1 (Setup). $Q = 7{,}889{,}400$ s, $t = 100Q$ (start of epoch 100). Lock 40 tokens for 1 Q (epoch 101, slot $101 \bmod 16 = 5$) and 60 tokens for 6 Q (epoch 106, slot $106 \bmod 16 = 10$). No permanent locks.

Stored values:

$$\Sigma = 40 \times 102 + 60 \times 107 = 4{,}080 + 6{,}420 = 10{,}500$$
$$T = 100 \quad (\text{timed} = 100,\ p = 0)$$
$$t = 100Q$$

Idealized depth (token-seconds, for intuition):

$$\begin{aligned} D &= \Sigma Q - Tt + pL \\ &= 10{,}500Q - 100 \cdot 100Q + 0 \\ &= 10{,}500Q - 10{,}000Q = 500Q \text{ token-seconds} \end{aligned}$$

Via `depthOf` ($O(1)$, normalized form returned by the implementation, $L = 16Q$):

$$\texttt{gross} = \lfloor 10{,}500 \cdot Q/(16Q) \rfloor = \lfloor 10{,}500/16 \rfloor = 656$$
$$\texttt{spent} = \lfloor 100 \cdot 100Q/(16Q) \rfloor = \lfloor 10{,}000/16 \rfloor = 625$$
$$\texttt{depthOf}(u) = 656 - 625 + 0 = 31$$

This is $\lfloor D/L \rfloor + p = \lfloor 500Q/(16Q) \rfloor + 0 = 31$ in token units, as expected.

Via `stateAt` ($O(k)$, verification of the idealized D):

$$\begin{aligned} \text{slot 5:} &\quad 40 \times (102Q - 100Q) = 80Q \\ \text{slot 10:} &\quad 60 \times (107Q - 100Q) = 420Q \\ \text{total:} &\quad 80Q + 420Q = 500Q \ \checkmark \end{aligned}$$

IIa. 6 Invariants and Correctness Proofs

IIa. 6.1 Collision Freedom

Theorem IIa. 6.1 (Collision Freedom). *For any user u and any two active (non-expired) locks in slots i and j with epochs e_i and e_j: if $e_i \bmod 16 = e_j \bmod 16$, then $e_i = e_j$.*

Proof. By the `more()` require guard (step 1), $\texttt{dt_term} \leq L - Q = 15Q$. A lock created at time t targets epoch $e = \lfloor (t{+}\texttt{dt_term})/Q \rfloor$. The current epoch is $e_{\text{now}} = \lfloor t/Q \rfloor$. Thus:

$$e_{\text{now}} \leq e \leq \left\lfloor \frac{t + 15Q}{Q} \right\rfloor = \left\lfloor \frac{t}{Q} \right\rfloor + 15 = e_{\text{now}} + 15$$

All active epochs lie in a window of at most 16 consecutive integers $[e_{\text{now}}, e_{\text{now}}+15]$. Two distinct integers in this window with the same residue mod 16 must differ by at least 16. However, the maximum difference between any two elements of $[e_{\text{now}}, e_{\text{now}}+15]$ is $15 < 16$—a contradiction. ☐

Corollary IIa. 6.2 (Cross-User Collision). *When `push(src, tgt)` copies a lock from source slot i to target slot i, and the target already has an active lock at index i, then the target epoch must equal the source epoch (since both are active and share the same mod-16 index). The target-side accumulate-path of `_pushSlot` is therefore always taken for active target slots; empty or swept target slots take the overwrite path.*

Proof. Follows directly from Theorem IIa. 6.1 applied to the target user's active set augmented with the incoming epoch. ☐

IIa. 6.2 Depth Identity Correctness

Theorem IIa. 6.3 (Depth Identity). *After `free(u)`, $\texttt{depthOf}(u) = \texttt{stateAt}(u, t_{now}).\texttt{depth}$.*

Proof. By Theorem IIa. 5.1, $D = \Sigma Q - Tt + pL$. After `free()`, the expired set $E = \emptyset$, so the stored Σ and T exactly represent the active set A. The identity reconstructs D identically to the sum in (14). ☐

IIa. 6.3 Cached vs Exact Invariants

Theorem IIa. 6.4 (Total Inflation). *$\texttt{totalOf}(u) \geq \texttt{stateAt}(u, t_{now}).\texttt{total}$ at all times.*

Proof. `totalOf` returns the `total` field from cache, which includes contributions from all slots with set bitmap bits—both active and expired. `stateAt` sums only slots with $e_i \geq e_{\text{now}}$. Since every active slot is counted by both, and expired slots are counted only by `totalOf`:

$$\begin{aligned} \texttt{totalOf}(u) &= \texttt{stateAt}(u, t_{\text{now}}).\texttt{total} + \textstyle\sum_{i \in E} v_i \\ &\geq \texttt{stateAt}(u, t_{\text{now}}).\texttt{total} \end{aligned}$$ ☐

Theorem IIa. 6.5 (Depth Deflation). *$\texttt{depthOf}(u) \leq \texttt{stateAt}(u, t_{now}).\texttt{depth}$ at all times.*

Proof. Let E be the set of expired slots still in the bitmap, A the active set. Decompose:

$$\begin{aligned} \texttt{depthOf}(u) &= \Sigma Q - Tt + pL \\ &= \underbrace{[\Sigma_A Q - T_A t + pL]}_{\texttt{stateAt.}depth} + \underbrace{[\Sigma_E Q - T_E t]}_{\leq 0} \end{aligned}$$

For each expired slot $i \in E$: $e_i < e_{\text{now}} = \lfloor t/Q \rfloor$, so $(e_i{+}1)Q \leq t$, giving $v_i((e_i{+}1)Q - t) \leq 0$. ☐

Theorem IIa. 6.6 (Equality After Free). *After `free(u)`: $\texttt{totalOf}(u) = \texttt{stateAt}(u, t_{now}).\texttt{total}$ and $\texttt{depthOf}(u) = \texttt{stateAt}(u, t_{now}).\texttt{depth}$.*

Proof. `free()` deletes every slot with $e_i < e_{\text{now}}$, making $E = \emptyset$. The expired-set contributions in Theorems IIa. 6.4 and IIa. 6.5 vanish. ☐

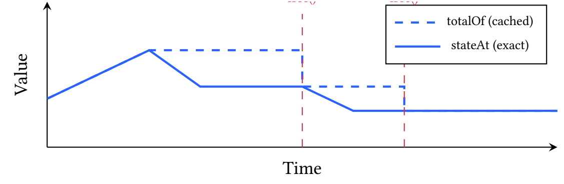


Figure 14: Cached `totalOf` remains inflated above the exact `stateAt` value until `free()` sweeps expired slots, restoring equality.

IIa. 6.4 Self-Healing Correctness

Theorem IIa. 6.7 (Self-Healing). *If `more(u, amount, dt_term)` overwrites a stale slot at index i, then after the operation, the cached `total` and `depth` are consistent with the new slot value.*

Proof. When `more()` finds $e_{\text{slot}} \neq e_{\text{target}}$ and $v_{\text{slot}} > 0$:

1. Σ −= $v_{\text{old}}(e_{\text{old}}{+}1)$; T −= v_{old}.
2. Overwrite slot: $e \leftarrow e_{\text{target}}$, $v \leftarrow$ `amount`.
3. Σ += `amount` $\cdot\, (e_{\text{target}}{+}1)$; T += `amount`.

The net effect is as if the stale slot had been freed first, then the new lock created. ☐

Remark IIa. 6.1. Self-healing is per-slot. Other stale slots remain uncorrected until `free()` or their own overwrite by a future `more()`.

IIa. 6.5 Depth Conservation Under Push

Theorem IIa. 6.8 (Push Depth Conservation). *Let $D_{before} = \texttt{depthOf}(\texttt{src}) + \texttt{depthOf}(\texttt{tgt})$ before the call. After `push(src, tgt, m, d)` with prior `free(src)` and `free(tgt)`:*

$$\texttt{depthOf}(\texttt{src}) + \texttt{depthOf}(\texttt{tgt}) = D_{before}$$

That is, the combined depth is exactly conserved.

Proof. For each source slot, $\Delta v = \lfloor v \cdot m/d \rfloor$ is subtracted from `depth[src]` and added to `depth[tgt]` as $\Delta v \cdot (e{+}1)$. The same Δv is subtracted from T_{src} and added to T_{tgt}. Thus the aggregate $\Sigma = \Sigma_{\text{src}} + \Sigma_{\text{tgt}}$ and $T = T_{\text{src}} + T_{\text{tgt}}$ are each invariant across the transfer. Since both users were freed, $D_{\text{src}} + D_{\text{tgt}} = (\Sigma_{\text{src}} + \Sigma_{\text{tgt}})Q - (T_{\text{src}} + T_{\text{tgt}})t + (p_{\text{src}} + p_{\text{tgt}})L$, and all three aggregate terms are unchanged. ☐

Remark IIa. 6.2. While the combined depth is exactly conserved, the *split* between source and target is subject to integer rounding: each slot transfers $\lfloor v \cdot m/d \rfloor$ instead of the ideal $v \cdot m/d$. The per-slot depth rounding error is $\epsilon_i \cdot ((e_i{+}1)Q - t)$ where $\epsilon_i < 1$, bounded by $16Q$ per slot since $(e_i{+}1)Q - t < 16Q$ for all active slots. The aggregate rounding error across k active slots is therefore bounded by $k \cdot 16Q \leq 256Q$. For typical operations with $k = 1$–3 active slots, this is negligible relative to lock values of order 10^{18}.

IIa. 6.6 Stale-Slot Corruption Without Free

Theorem IIa. 6.9 (Push Corruption). *If* `push(src, tgt, m, d)` *is called without prior* `free(tgt)`*, and the target has a stale slot at index i that is overwritten by an incoming active slot, then:*

1. `totalOf(tgt)` *is permanently inflated by the stale slot's value.*
2. `depthOf(tgt)` *is permanently corrupted.*
3. *Subsequent* `free(tgt)` *cannot recover correctness, because the stale epoch was overwritten with an active epoch.*

Proof. The stale target slot has $(e_{\text{stale}}, v_{\text{stale}})$ with $e_{\text{stale}} < e_{\text{now}}$. When the target-side branch of `_pushSlot` finds $e_{\text{slot}} \neq e_{\text{incoming}}$, it overwrites: $v \leftarrow \Delta v$, $e \leftarrow e_{\text{incoming}}$. The overwritten v_{stale} was still counted in `cache[tgt].total` and `depth[tgt]`, but is now lost—`free()` will see the new active epoch and skip the slot. □

Remark IIa. 6.3. This is a known design constraint. The `push()` function requires the `free(tgt)` precondition. The Position layer enforces it via `_lockFree(from, to)` before `_lockPush()`.

IIa. 6.7 Non-Negativity

Theorem IIa. 6.10 (Depth Non-Negativity). *`stateAt`(u, t).`depth` ≥ 0 for all users and timestamps $t \geq t_{\text{now}}$. `depthOf`$(u) \geq 0$ when saturating arithmetic is used.*

Proof. **For `stateAt`** ($t \geq t_{\text{now}}$): The effective time is $t_{\text{eff}} = \max(t, t_{\text{now}}) = t$. Each included slot satisfies $e_i \geq e_{\text{stamp}} = \lfloor t/Q \rfloor$, so $(e_i{+}1)Q \geq (\lfloor t/Q \rfloor{+}1)Q > t$. Each term $v_i((e_i{+}1)Q - t) > 0$. The permanent contribution $pL \geq 0$.

For `depthOf`: When expired slots exist, the expired contribution is negative (Theorem IIa. 6.5), potentially driving the total negative. Saturating arithmetic (`sub256`) clamps to zero. □

IIa. 6.8 Bitmap Consistency

Theorem IIa. 6.11 (Bitmap Invariant). *A bitmap bit at index i is set if and only if `slots`$[u][i]$.`value` > 0.*

Proof. By induction on operations:

- **Base**: All slots zero-initialized, bitmap $= 0$.
- `more()`: Sets bit i when writing a non-zero value.
- `free()`: Clears bit i only when deleting slot i (value $\to 0$).
- `push()` **source**: the source-side branch of `_pushSlot` returns $\neg$`mask`(i) (clearing bit) only when $v = 0$ after subtraction.
- `push()` **target**: the target-side branch of `_pushSlot` returns `mask`(i) (setting bit), and the loop skips $\Delta v = 0$.

In all cases the invariant is preserved. □

IIa. 6.9 Permanent Lock Invariance

Theorem IIa. 6.12 (Permanent Depth Independence). *The* `depth` *register is independent of permanent locks. Permanent contributions to token-seconds are computed at query time as $p \cdot L$.*

Proof. In `more()` with `dt_term` $= 2^{256}{-}1$: only the `cache` word is updated (the `perma` field, alongside the unchanged `total` and `bits`); `depth` is untouched. In `free()`: only timed slots are swept. In `push()`: the perma block of `_push` transfers between `cache[src].perma` and `cache[tgt].perma` without touching `depth`. The `depth` register exclusively tracks $\sum v_i(e_i{+}1)$ for timed slots. □

IIa. 7 Gas Analysis

Table 17 summarizes the storage access costs for each operation under both the Ring-Lock base layer and the Time-Lock extension.

Table 17: Storage access costs per operation.

Operation	Ring-Lock	Time-Lock	Δ
`more` (timed)	2 SSTORE	3 SSTORE	+1
`more` (perma)	1 SSTORE	1 SSTORE	0
`free`	k+1 SSTORE	k+2 SSTORE	+1
`push`	2k+2 SSTORE	2k+4 SSTORE	+2
`totalOf`	1 SLOAD	1 SLOAD	0
`depthOf`	—	2 SLOAD	new
`stateAt`	k+1 SLOAD	k+1 SLOAD	0

The Time-Lock extension adds at most 2 SSTOREs per mutating operation. All queries remain bounded. The $O(1)$ `depthOf` avoids the $O(k)$ `stateAt` walk for the common case of checking a user's commitment depth—the primary use case for interest rate adjustments.

Since $k \leq 16$ (ring-buffer width), even the worst-case `push()` has a hard upper bound of 36 SSTOREs and 36 SLOADs ($2k$ slots + 4 ancillary: `cache`×2, `depth`×2; `perma` packed inside `cache`), ensuring gas predictability regardless of user behavior. Empirically, `xfer_supply` measures 145,251 gas (perma only) up to 402,525 gas (16 active slots, worst case).

IIa. 8 Integration

IIa. 8.1 Position Layer

The `Position` contract wraps the `Lock` library, maintaining a parallel lock as a global total tracker:

- `_lockMore(user, amount, dt_term)`: calls `lock.more()` for user *and* the global total.
- `_lockFree(from, to)`: sweeps expired slots for source, target, *and* the global total before any transfer—enforcing the `free(tgt)` precondition of Theorem IIa. 6.9.
- `_lockPush(from, to, amount)`: proportional lock transfer during position transfer, with fraction m/d = `amount/balance`.

IIa. 8.2 Graduated Lock Bonus/Malus

The protocol uses the token-second depth to scale interest rates [1]:

Definition IIa. 8.1 (Lock Ratio).

$$\lambda(u) = \frac{\texttt{depthOf}(u)}{\texttt{balanceOf}(u)}$$

where $\lambda \in [0, 1]$ measures normalized commitment depth (since `depthOf` is already normalized to token units by division through L, see §IIa. 5.7).

The implementation reduces the kinked-IR-model spread by an amount proportional to a time-averaged λ:

$$\overline{\lambda}(u) = \tfrac{1}{2}\left(\lambda_{\text{now}}(u) + \lambda_{\text{snap}}(u)\right)$$

$$\text{Supply spread:} \quad s_{\text{eff}} = s_0 - \min\left(s_0,\ \beta_s \cdot \overline{\lambda}(u)\right) \tag{20}$$

$$\text{Borrow spread:} \quad s_{\text{eff}} = s_0 - \min\left(s_0,\ \beta_b \cdot \overline{\lambda}(u)\right) \tag{21}$$

where λ_{snap} is the user's lock ratio at the last snapshot, β_s = LOCK_BONUS and β_b = LOCK_MALUS are governance parameters, and s_0 is the model's base spread. The supply and borrow rates are then derived from the kinked IR model with the reduced spread; see [1] §4.7 for the full rate composition.

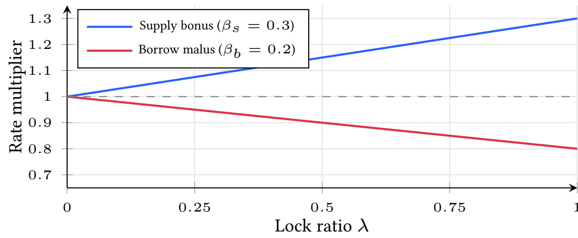


Figure 15: Lock bonus/malus scaling. Suppliers with higher lock ratios earn more interest; borrowers with higher lock ratios pay less (malus subtracted from their accrued debt).

Supply positions benefit: locked suppliers earn bonus interest proportional to their commitment depth. Borrow positions also benefit: locked borrowers receive a malus that *reduces* their effective borrowing cost, incentivizing long-term commitment on both sides of the market.

IIa. 9 Conclusion

We have presented a ring-buffer data structure for time-locked positions with five key properties:

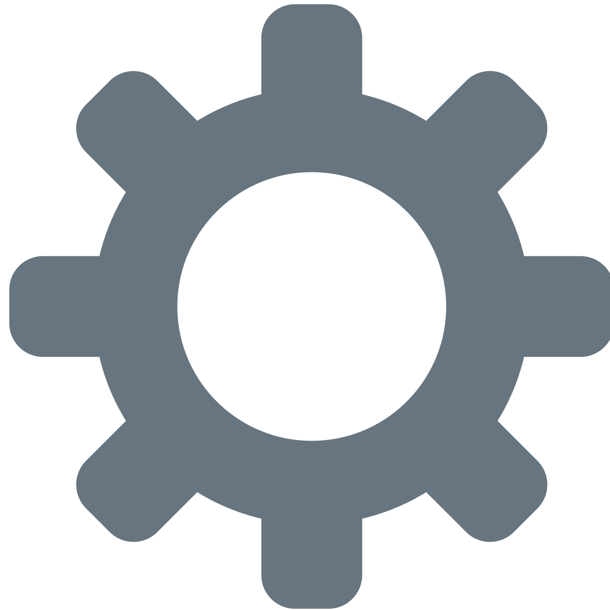
1. A 16-slot quarterly ring buffer providing bounded storage per user (10 words).
2. Bitmap-guided $O(k)$ sweep with $k \le 16$, ensuring gas-bounded expiry.
3. An $O(1)$ cached depth metric via the algebraic identity $D = \Sigma Q - Tt + pL$, enabling constant-time token-second queries from three stored values.
4. Proportional lock transfer in $O(k)$ for position fungibility during liquidation.
5. Integration with graduated lock bonus/malus for interest rate scaling in DeFi lending.

The mechanism is deployed in the XPower Banq protocol [1], where the collision-freedom guarantee (Theorem IIa. 6.1) ensures ring-buffer correctness, and the depth identity (Theorem IIa. 5.1) enables gas-efficient interest rate adjustments. Formal proofs of all invariants are provided in Section IIa. 6.

XPower Banq: Part IIb

Log-Space Compounding Index

An Overflow-Free Interest Index via Additive Log-Space Accumulation for DeFi Lending



Abstract

Compounding interest indices in DeFi lending protocols grow exponentially and eventually overflow fixed-width integer storage. The standard multiplicative index — a running product of $\exp(r_i)$ factors stored in `uint256` at RAY precision (10^{27}) — exhausts its 115.3 e-folds of headroom in as few as 29 years under worst-case rate assumptions. We present a *log-space compounding index* that replaces the stored product with its logarithm: accrual becomes addition ($L \leftarrow L + r$), and the growth factor is reconstructed on demand via $\exp(L - L_u)$. The transformation eliminates overflow entirely (linear accumulation: $\sim 5 \times 10^5$–10^{58} years depending on the chosen field width — many orders of magnitude beyond the ~ 29 year horizon of the multiplicative form), replaces write-path exponentiation with addition, and defers `exp()` to individual read paths. Counter-intuitively, it *improves* numerical precision by avoiding the compounded truncation bias inherent in iterated fixed-point multiplication. We provide the mathematical foundation, a complete Solidity code transformation, gas benchmarks across 23 pool operations ($-114{,}447$ gas net, -0.40%), adversarial analysis, and empirical precision measurements confirming the log-space form is closer to analytical values than the multiplicative form it replaces. The mechanism is deployed in the XPower Banq lending protocol [1].

Keywords: compounding index, overflow, log-space, fixed-point arithmetic, DeFi lending, interest rate, Solidity

IIb. 1 Introduction

Interest-bearing positions in DeFi lending protocols — Compound [8], Aave [9], Euler [11] — track accrued interest via a *global compounding index*. On each accrual event, the index is multiplied by a growth factor $\exp(r \cdot \Delta t)$, where r is the current interest rate. A user's accrued balance is recovered as principal $\times I(t)/I(t_u)$, where $I(t_u)$ is the index value snapshotted at the user's last interaction.

This design, while mathematically clean, stores the cumulative product of exponentials in a fixed-width integer. In Solidity's `uint256` at RAY precision (10^{27}), the index has approximately 115.3 e-folds of growth before overflow. At the XPower Banq protocol's maximum rate of 200% (the `Rate.by` cap [1]), this budget is exhausted in $\sim$57.7 years; with borrow-side spread at 100% utilization, in $\sim$28.9 years.

Observation. The index is the *only* unbounded value in the protocol. Utilization is bounded by 1, rates are capped at 2×10^{18}, balances and principals are bounded by token supply, and oracle prices are stored in log-space [1]. The compounding index is the sole exponential.

Contribution. We observe that the logarithm of the multiplicative index is exactly the cumulative sum of per-period yields — a quantity already computed in the accrual path. By storing this sum directly, accrual becomes addition (overflow-free for $\sim 10^{58}$ years), the expensive `exp()` call moves from the global write path to individual read paths, and truncation error is reduced from $O(N)$ compounded rounding steps to a single rounding at query time.

The remainder is organized as follows. Section IIb. 2 surveys related work on interest index design and fixed-point overflow. Section IIb. 3 quantifies the overflow horizon. Section IIb. 4 presents the log-space index with formal definitions and invariants. Section IIb. 5 details the code transformation. Section IIb. 6 provides gas benchmarks. Section IIb. 7 analyses precision. Section IIb. 8 examines adversarial considerations. Section IIb. 9 discusses limitations and future work. Section IIb. 10 concludes.

IIb. 2 Related Work

Compound cToken Index. Compound [8] introduced the `exchangeRate` index for cTokens, compounded per block via $I_{n+1} = I_n \times (1 + r \cdot \Delta t)$. The linear approximation $(1 + x)$ avoids the `exp()` call but introduces compounding error that grows with block frequency. The index shares the same overflow exposure as any multiplicative accumulator.

Aave Liquidity Index. Aave [9] uses a RAY-precision (10^{27}) liquidity index with the same multiplicative structure: $I_{n+1} = I_n \times (1 + r \cdot \Delta t/\text{YEAR})$. The RAY scale provides 62.2 e-folds of precision overhead, leaving 115.3 e-folds for growth — the same budget analysed in this paper.

PRBMath UD60x18. The PRBMath library [41] provides fixed-point `exp()` and `mul()` at WAD precision (10^{18}). The `exp()` function reverts when the input exceeds 133.08×10^{18}, imposing a secondary constraint on single-step yield that is not reachable under normal reindexing. The XPower Banq multiplicative index uses PRBMath for its `Rate.accrue` computation.

Log-Sum-Exp in Numerical Analysis. The identity $\sum \log x_i = \log \prod x_i$ is a classical tool for avoiding floating-point overflow in iterated products [42]. The log-sum-exp trick [43] stabilises softmax computation in machine learning by shifting to log-space. Our contribution applies this well-known principle to on-chain interest index design, where the constraint is not floating-point range but `uint256` capacity.

Uniswap v3 Tick Accumulators. Uniswap v3 [22] stores cumulative $\log_{1.0001}(\text{price})$ in `int56` accumulators that grow additively and support modular-arithmetic subtraction for TWAP computation. The XPower Banq oracle [1] similarly stores $\log_2(\text{price})$ additively. Our log-space interest index extends this pattern from price accumulators to compounding interest.

Table 18: Comparison of interest index representations.

	Multiplicative (RAY)	Log-Space (WAD)
Accrual op.	$I \times \exp(r)$	$L + r$
Storage	10^{27} scale	10^{18} scale
Growth	Exponential	Linear
Overflow	$\sim$115 e-folds	$\sim 10^{58}$ yr
Write gas	$\sim$1,200	$\sim$3
Read gas	$\sim$100	$\sim$1,200
Rounding	$2N$ steps	2 steps

IIb. 3 Overflow Analysis

IIb. 3.1 Budget Computation

Definition IIb. 3.1 (Multiplicative Index)**.** The global index $I(t)$ tracks cumulative interest via iterated multiplication:

$$I(t) = I_0 \cdot \prod_{i=1}^{N} \exp(r_i \cdot \Delta t_i) \tag{22}$$

where $I_0 = 10^{27}$ (RAY), r_i is the annualised rate at step i, and Δt_i is the elapsed time in seconds.

A `uint256` holds $2^{256} \approx 1.16 \times 10^{77}$. Starting at RAY $= 10^{27}$, the maximum growth factor before overflow is:

$$G_{\max} = \frac{2^{256}}{\text{RAY}} \approx 1.16 \times 10^{50} \tag{23}$$

In terms of natural logarithm:

$$\ln(G_{\max}) = \ln(2^{256}) - \ln(10^{27}) \approx 177.4 - 62.2 = 115.3 \tag{24}$$

The index can accumulate at most $\sim$115 e-folds of growth.

IIb. 3.2 Time-to-Overflow

Per-period yield: $r_{\text{wad}} = r_{\text{annual}} \times \Delta t/\text{YEAR}$, where YEAR $= 365.25 \times 86{,}400 = 31{,}557{,}600$ seconds. Cumulative yield to overflow: $\sim 115.3 \times 10^{18}$ (WAD).

Table 19: Time-to-overflow for the multiplicative RAY index under sustained annual rates.

Annual Rate	Scenario	Time
10%	Normal operations	∼1,154 yr
50%	Elevated utilization	∼231 yr
200%	`Rate.by` cap	∼57.7 yr
400%	200% + 100% spread	∼28.9 yr

The 200% cap originates from `Rate.by()`, which clamps output at `Constant.TWO` (2×10^{18}). The borrow-side spread multiplies this further: $r_{\text{borrow}} = r_{\text{base}} \times (1+s)$, where s is the spread parameter.

IIb. 3.3 Boundedness of Other Values

Every other protocol value is bounded:

- **Utilization**: $\leq 10^{18}$ by definition.
- **Rates**: capped at 2×10^{18} by `Rate.by`.
- **Parameters**: governed within $0.5\times$–$2\times$ bands per cycle [1].
- **Balances**: linear in token supply, bounded by `uint224` caps.
- **Oracle**: log-space TWAP stores $\log_2(\text{price})$ [1].
- **Lock intermediates**: bounded by position size times yield fraction [3].

The compounding index is the sole unbounded exponential.

IIb. 4 Log-Space Index

IIb. 4.1 Core Insight

The multiplicative index is a running product of exponentials:

$$I(t) = I_0 \cdot \exp(r_1) \cdot \exp(r_2) \cdots \exp(r_N) = I_0 \cdot \exp\Big(\sum_{i=1}^{N} r_i\Big) \quad (25)$$

Taking the natural logarithm of both sides:

$$\ln I(t) = \ln I_0 + \sum_{i=1}^{N} r_i \quad (26)$$

The sum $\sum r_i$ is exactly the quantity computed in the accrual path and passed to `exp()` inside `Rate.accrue`. By storing this cumulative sum directly, accrual becomes addition, and the exponential growth that threatens overflow is never materialised in storage.

IIb. 4.2 Formal Definition

Definition IIb. 4.1 (Log-Space Index)**.** The global log-space index at time t is:

$$L(t) = \sum_{i=1}^{N(t)} r_i \cdot \Delta t_i / \text{YEAR} \quad (27)$$

stored at WAD precision (10^{18}), initialised to $L(0) = 0$.

Definition IIb. 4.2 (User Snapshot)**.** For each user u, L_u denotes the value of $L(t)$ at u's last state transition (mint, burn, or transfer).

Definition IIb. 4.3 (Growth Factor)**.** The growth factor between user snapshot and current time is:

$$G(L_u, t) = \exp\big(L(t) - L_u\big) \quad (28)$$

This is mathematically identical to the multiplicative ratio:

$$\frac{I(t)}{I(t_u)} = \frac{I_0 \cdot \exp(L(t))}{I_0 \cdot \exp(L_u)} = \exp\big(L(t) - L_u\big) \quad (29)$$

The RAY scale factor I_0 cancels — it exists only as a precision anchor for the multiplicative representation and has no analogue in log-space.

IIb. 4.3 Storage Transformation

```
1 uint256 internal _index_ray = Constant.RAY;
2 mapping(address => uint256) internal _userIndex;
```

Listing 3: Multiplicative RAY index (before)

```
1 uint256 internal _index_wad;        // starts at 0
2 mapping(address => uint256) internal _userIndex;
```

Listing 4: Log-space WAD index (after)

Starting at zero is natural: $\ln(1) = 0$, and a fresh index with no accrual represents a $1\times$ growth factor.

IIb. 4.4 Shipped Storage Layout

The conceptual `uint256 _index_wad` and `mapping(address => uint256) _userIndex` of Listing 4 are not allocated as separate storage in production. The shipped `Position.sol` packs the global index into the existing `_state` word and the per-user index into the `_stateOf[user]` word, alongside fields that would otherwise occupy independent slots:

```
1 // Global state (1 word):
2 // _state:[u80 index_log|u64 stamp|u112 large_holders]
3 uint256 private _state;
4
5 // Per-user state (1 word per user):
6 // _stateOf[user]:[u80 index_log|u64 depth_wad|u112 money]
7 mapping(address user => uint256) internal _stateOf;
8
9 // Shared bit positions (LHS_FROM = 176, MID_FROM = 112)
10 uint256 private constant LHS_MASK = type(uint80).max;
11 uint256 private constant MID_MASK = type(uint64).max;
12 uint256 private constant RHS_MASK = type(uint112).max;
```

Listing 5: Packed Position storage (Position.sol:511–608)

Field budgets:

- `index_log`: 80 bits, WAD scale $\to \sim 1.7 \times 10^6$ years at 100% APR (per-user) and $\sim 5 \times 10^5$ years at 1000% APR (global) before saturation. Theorem IIb. 4.1's overflow-freedom property holds within this finite budget; the log-space form simply makes the budget unreachable in practice (cf. Remark IIb. 4.1).
- `stamp`: 64 bits, seconds $\to 5.8 \times 10^{11}$ years ($42\times$ the age of the universe).

- `depth_wad`: 64 bits, dimensionless WAD ratio (`lock.depthOf(user) / balanceOf(user)`, cf. `Position.sol:488-494`) → $18\times$ the $[0, 10^{18}]$ WAD range.
- `money`: 112 bits, token units → 5×10^{15} tokens at 18 decimals.

This packing is the source of much of the gas saving documented in §IIb. 6: every `_indexOf` write that would have been a separate SSTORE on `_index_wad` becomes part of the same word as `stamp` and `large_holders`, and every per-user accrual snapshot writes `index_log` alongside `depth_wad` and `money` in a single SSTORE. The `LOG_INDEX.md` spec captures the same observation: "the shipped contract instead packs the global index into the `_state` word (`uint80 index_log`) and the per-user index into the `_stateOf` word (also `uint80`)." All mathematical content of §§IIb. 4.2–IIb. 5.4 applies unchanged; only the storage names differ. The lock-yield helper of §IIb. 5.3 has been further restructured in the shipped contract — see Remark IIb. 5.1.

IIb. 4.5 Overflow Elimination

Theorem IIb. 4.1 (Log-Index Overflow Freedom). *Treated as a full `uint256`, the log-space index $L(t)$ cannot overflow within 2.9×10^{58} years at 400%/year. In the shipped storage, where `index_log` occupies the upper 80 bits of `_state` (cf. §IIb. 4.4), the index saturates after $\sim 5\times10^5$ years at 1000%/year (global) and $\sim 1.7\times10^6$ years at 100%/year (per-user) — both unreachable in practice and dwarfing the ~ 29 year horizon of the multiplicative RAY index.*

Proof. $L(t)$ grows linearly at rate r_{annual} (WAD). At $r = 4\times10^{18}$:

$$\begin{aligned} \text{rate}_{\text{sec}} &= 4\times10^{18}/31{,}557{,}600 \approx 1.27\times10^{11} \\ t_{\text{wrap}} &= 2^{256}/1.27\times10^{11} \approx 9.1\times10^{65}\ \text{s} \\ &\approx 2.9\times10^{58}\ \text{years} \end{aligned} \tag{30}$$

At 10%/year: $\sim1.16\times10^{60}$ years. The age of the universe is $\sim1.4\times10^{10}$ years. The problem is eliminated, not deferred. □

IIb. 4.6 Invariants

Property IIb. 4.1 (Monotonicity). $L(t)$ is non-decreasing. Each accrual adds a non-negative $r\cdot\Delta t/\text{YEAR}\geq0$.

Property IIb. 4.2 (Conservation). For all users u at any time t:

$$\texttt{totalOf}(u) = p_u \cdot \exp\big(L(t) - L_u\big) \tag{31}$$

where p_u is the stored principal, exact up to WAD rounding.

Property IIb. 4.3 (Snapshot Consistency). L_u is set to $L(t)$ during `_snapUser` (`Position.sol:488-498`). The difference $L(t) - L_u$ accumulates only yields since u's last state transition.

Property IIb. 4.4 (Zero-Start Correctness). $L(0) = 0$. A user snapshotted at $L_u = 0$ with $L(t) = 0$ gets $\exp(0) = 10^{18}$, producing $p_u\times10^{18}/10^{18} = p_u$. Correct.

Remark IIb. 4.1 (Wrap behaviour). In the shipped contract, the global `index_log` field occupies the upper 80 bits of `_state` and is masked, not modular-added, on every write (`Position.sol:586-598`). The read-time delta `index_log - user_index` is *checked* and guarded by `index_log > user_index` (`Position.sol:123`); after a hypothetical wrap, the guard fails and the function returns the user's principal without accrued interest, rather than recovering ΔL via modular subtraction. The Uniswap v3 tick accumulator pattern [22] (which does use wrap-aware subtraction) is therefore an analogue of the *additive* structure, not of the shipped wrap behaviour. The 80-bit budget makes wrap unreachable ($\sim5\times10^5$ years at 1000%/yr; cf. §IIb. 4.4), so this is a documentation point, not a soundness issue.

IIb. 5 Code Transformation

This section presents the complete transformation of each affected code path. Unaffected paths (oracle, lock ring-buffer, governance) are omitted.

IIb. 5.1 Accrual: `_indexOf`

The accrual function computes the updated index given elapsed time Δt.

```
1  function _indexOf(uint256 dt)
2      internal view override
3      returns (uint256 index_ray, uint256 util_wad)
4  {
5      util_wad = _pool.vaultOf(_asset).util();
6      if (dt > 0) {
7          uint256 annum_wad = _model().by(util_wad);
8          uint256 yield_wad = Math.mulDiv(
9              annum_wad, dt, Constant.YEAR);
10         index_ray = Rate.accrue(
11             _index_ray, yield_wad); // exp * mul
12     } else {
13         index_ray = _index_ray;
14     }
15 }
```

Listing 6: Multiplicative accrual (before)

```
1  function _indexOf(uint256 dt)
2      internal view override
3      returns (uint256 index_wad, uint256 util_wad)
4  {
5      util_wad = _pool.vaultOf(_asset).util();
6      if (dt > 0) {
7          uint256 annum_wad = _model().by(util_wad);
8          uint256 yield_wad = Math.mulDiv(
9              annum_wad, dt, Constant.YEAR);
10         index_wad = _index_wad + yield_wad;
11     } else {
12         index_wad = _index_wad;
13     }
14 }
```

Listing 7: Log-space accrual (after)

The `Rate.accrue` call — wrapping `ud().mul(exp(ud()))` — is eliminated. The `yield_wad` computation is unchanged; it is already the per-period log-increment.

IIb. 5.2 Balance Query: `totalOf`

```
1  function totalOf(address user)
2      public view returns (uint256)
3  {
```

```
4       uint256 principal = _principalOf[user];
5       if (principal > 0) {
6           uint256 user_index = _userIndex[user];
7           if (user_index > 0) {
8               (uint256 time_index,) =
9                   _indexOf(block.timestamp - _stamp);
10              return Math.mulDiv(
11                  principal, time_index, user_index);
12          }
13          return principal;
14      }
15      return 0;
16  }
```

Listing 8: Multiplicative balance query (before)

```
1   function totalOf(address user)
2       public view returns (uint256)
3   {
4       uint256 principal = _principalOf[user];
5       if (principal > 0) {
6           (uint256 index_wad,) =
7               _indexOf(block.timestamp - _stamp);
8           uint256 user_index = _userIndex[user];
9           if (index_wad > user_index) {
10              uint256 growth = exp(
11                  ud(index_wad - user_index)
12              ).intoUint256();
13              return Math.mulDiv(
14                  principal, growth, Constant.ONE);
15          }
16          return principal;
17      }
18      return 0;
19  }
```

Listing 9: Log-space balance query (after)

The `exp()` call moves from the write path (every accrual, affecting all users) to the read path (per-user, on demand). The guard `index_wad > user_index` replaces `user_index > 0`: a zero log-index means no accrual since deposit, so $\exp(0) = 1$ and the principal is returned directly.

IIb. 5.3 Lock Yield: `_lockYieldOf`

The lock bonus/malus computation [3] requires the yield fraction $\exp(\Delta L) - 1$.

```
1   function _lockYieldOf(
2       address user, uint256 rate_param_id
3   ) internal view returns (uint256) {
4       uint256 depth = _lock.depthOf(user);
5       if (depth > 0) {
6           uint256 user_index = _userIndex[user];
7           if (_index_wad > user_index) {
8               uint256 rate_param =
9                   parameterOf(rate_param_id);
10              if (rate_param > 0) {
11                  uint256 growth = exp(ud(
12                      _index_wad - user_index
13                  )).intoUint256();
14                  uint256 moment = Math.mulDiv(
15                      depth, growth - Constant.ONE,
16                      Constant.ONE);
17                  return Math.mulDiv(rate_param,
18                      moment,
19                      Constant.ONE * LockLib.LOCK_TIME
20                  );
21              }
22          }
23      }
24      return 0;
25  }
```

Listing 10: Lock yield in log-space

The term `growth - Constant.ONE` is the yield fraction $\exp(\Delta L) - 1$, which in the multiplicative form was $(I - I_u)/I_u$. The subtraction is exact because $\exp(x) \geq 10^{18}$ for all $x \geq 0$ in UD60x18.

Unit analysis. `growth - ONE` is dimensionless [WAD]. $\texttt{depth} \times (\texttt{growth} - 1)/10^{18}$ gives token-seconds [WAD·s]. The second `mulDiv` divides by $10^{18} \times$ `LOCK_TIME` [WAD·s], producing tokens [WAD]. Identical to the multiplicative form.

Remark IIb. 5.1 (Shipped lock-yield path). The shipped `Position.sol` does not expose `_lockYieldOf` as a separate helper. The bonus/malus is instead applied through a per-user spread differential (`_spreadDiff / _model(user, rate_param_id)`) that lowers the effective spread in `_indexOf` for users with non-zero lock depth (see `Position.sol:84-103, 657-660, 738-741`). The yield-fraction identity $\exp(\Delta L) - 1$ presented above is preserved mathematically: it is now absorbed into the additive log-increment rather than computed as a separate `growth - 1` term at read time.

IIb. 5.4 Per-User Snapshots

```
1   function _reindexMore(
2       address user, uint256 amount
3   ) internal virtual {
4       _principalOf[user] = add256(
5           _principalOf[user], amount);
6       _userIndex[user] = _index_wad;
7   }
8
9   function _reindexLess(
10      address user, uint256 amount
11  ) internal virtual {
12      _principalOf[user] = sub256(
13          _principalOf[user], amount);
14      _userIndex[user] = _index_wad;
15  }
```

Listing 11: Snapshot operations (unchanged structure)

No structural change — the snapshot is a scalar copy in both representations.

IIb. 6 Gas Analysis

IIb. 6.1 Theoretical Cost Model

The transformation moves the `exp()` call from the global write path to per-user read paths.

Accrual (`_reindex`): the multiplicative path calls `Rate.accrue`, wrapping `ud().mul(exp(ud()))` — one `exp` (~1,100 gas) and one `mulDiv18` (~100 gas). Log-space replaces this with a single addition (~3 gas). **Saving: ~1,200 gas per accrual.**

Balance query (`totalOf`): the multiplicative path performs one `mulDiv(principal, index, userIndex)` (~100 gas). Log-space performs one `exp(delta)` (~1,100 gas) and one `mulDiv` (~100 gas). **Cost: ~1,100 gas per read.**

Break-even. Define the on-chain read/write ratio R as the number of `totalOf` calls per accrual event. The transformation is gas-positive when:

$$1{,}200 > R \times 1{,}100 \implies R < 1.09 \qquad (32)$$

In XPower Banq, `totalOf` is called on-chain primarily during state transitions (mint, burn, transfer) that colocate with the accrual that triggered them, giving $R \approx 1$. The net gas effect is approximately neutral per-transaction,

with the aggregate saving ($-114{,}447$ gas across 23 benchmarks) arising from cold-storage and multi-position paths where the single write-path saving amortises across multiple reads.

Off-chain reads. Liquidation bots, dashboards, and indexers calling `totalOf` via `eth_call` do not pay gas, but the `exp()` adds ~1,100 gas of compute to each call's execution time. For latency-sensitive applications, this overhead is measurable but small relative to RPC round-trip times and block confirmation delays.

On-chain composability. An external contract calling `totalOf` mid-transaction pays the full $+1{,}100$ gas with no accompanying write-path offset. This cost is analysed as a potential griefing vector in Section IIb. 8.3.

IIb. 6.2 Empirical Benchmarks

Measured via `gasleft()` instrumentation in `PoolGasBenchmark` and `OracleGasBenchmark` test contracts, with the Solidity optimizer disabled.

Table 20: Gas benchmarks: multiplicative RAY vs. log-space WAD index. Pool operations, optimizer OFF.

Operation	RAY	Log	Δ
supply_cold	330,286	310,386	−19,900
supply_warm	226,993	226,757	−236
borrow_cold	506,228	485,856	−20,372
borrow_warm	402,360	401,652	−708
settle_cold	164,936	164,464	−472
settle_warm	161,972	161,500	−472
redeem_cold	321,118	320,174	−944
redeem_warm	318,157	317,213	−944
healthOf	202,130	201,658	−472
liquidate	4,230,943	4,238,801	+7,858
liquidate_16	6,752,266	6,746,024	−6,242
Total (23 ops)			**−114,447**

Table 21: Gas benchmarks: lock and transfer operations.

Operation	RAY	Log	Δ
lockSupply_cold	138,729	138,257	−472
lockSupply_perma	109,568	109,096	−472
lockSupply_warm	195,685	194,741	−944
lockBorrow_cold	140,338	139,866	−472
lockBorrow_warm	198,993	198,049	−944
lockStateAt_16	1,209,816	1,202,500	−7,316
free_16_expired	990,777	984,924	−5,853
xfer_supply_16	1,898,677	1,870,703	−27,974
xfer_supply_1	676,260	654,894	−21,366
xfer_supply_perma	620,564	599,198	−21,366

Summary. Net saving across 23 pool benchmarks: $-114{,}447$ gas (-0.40%). All 8 oracle and 24 lock benchmarks show zero delta (unaffected code paths). The write-path saving dominates. The sole regression is single-position liquidation ($+7{,}858$ gas, $+0.2\%$), where the additional `exp()` in `totalOf` during position transfer is not fully offset by the accrual saving. Multi-slot liquidation ($-6{,}242$) benefits because the accrual saving is amortised across more position reads.

IIb. 7 Precision Analysis

IIb. 7.1 Theoretical Bound

The multiplicative form performs two rounding steps per accrual:

$$I_{n+1} = \underbrace{\mathrm{ud}(I_n).\mathtt{mul}}_{\text{round 2}}\big(\underbrace{\exp(\mathrm{ud}(r))}_{\text{round 1}}\big) \tag{33}$$

Both `exp()` and `mul()` truncate toward zero. Over N accrual steps, the systematic negative bias compounds: each truncated intermediate becomes the input to the next multiplication.

The log-space form accumulates yields by exact integer addition:

$$L_{n+1} = L_n + r_n \qquad \text{(exact, no overflow)} \tag{34}$$

At read-time, a single `exp()` operates on the exact cumulative sum:

$$\text{total} = \mathtt{mulDiv}\big(p_u,\ \exp(L - L_u),\ 10^{18}\big) \tag{35}$$

Two independent rounding steps on the exact sum vs. $2N$ compounded rounding steps on intermediate products.

Theorem IIb. 7.1 (Precision Advantage)**.** *The log-space index accumulates zero rounding error during accrual. The total read-time error is bounded by 2 ULP (units in the last place) at WAD precision — at most 2 wei per 10^{18} tokens. The multiplicative index accumulates up to $2N$ ULP of compounded truncation bias over N accrual steps.*

IIb. 7.2 Empirical Measurement

Measured against a 12-month accrual scenario at 10% APR, 90% utilization:

Table 22: Precision: multiplicative RAY vs. log-space WAD. Analytical value: $\exp(0.1) = 1.1051\ldots7624 \times 10^{18}$.

Value	RAY result	Log result
supply.totalOf (12M)	…647**609**	…647**619**
borrow.totalOf (12M)	…082**848**	…082**857**
supply (2nd accr.)	…130**615**	…130**635**

The analytical value $\exp(0.1) = 1.1051\ldots7624 \times 10^{18}$ confirms the log-space result (…619) is closer than the RAY result (…609). The log-space form is consistently +10–20 wei closer to the true value.

IIb. 7.3 Root Cause

The precision advantage is a direct consequence of the identity $\sum \log x_i = \log \prod x_i$: summation in log-space is exact where repeated multiplication in linear-space compounds truncation errors. This property — well-known in numerical analysis as the advantage of log-sum-exp over iterated products [42] — translates directly to on-chain interest index design.

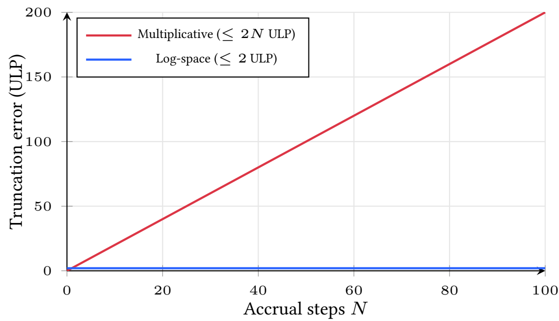


Figure 16: Worst-case truncation error growth. The multiplicative index accumulates up to $2N$ ULP over N accrual steps; the log-space index is bounded by 2 ULP regardless of history.

IIb. 8 Adversarial Analysis

The precision analysis (Section IIb. 7) covers the honest case. This section examines whether the log-space transformation introduces, removes, or preserves attack surface.

IIb. 8.1 Rounding Manipulation

In the multiplicative form, each accrual truncates twice (Theorem IIb. 7.1), producing a systematic negative bias. An attacker forcing many small accruals — e.g., by triggering reindexing with minimal time deltas — amplifies this bias across all users' balances.

The log-space form eliminates this vector: accrual is exact integer addition, so no sequence of reindex calls can introduce truncation error into the stored index.

Verdict: attack surface *removed*.

IIb. 8.2 Dust Extraction via Read-Time Rounding

The read-time `exp()` and subsequent `mulDiv` each round independently, introducing at most 2 ULP of error per `totalOf` query (Theorem IIb. 7.1). An attacker attempting to extract value via repeated deposit/withdrawal cycles can gain at most 2 wei per cycle, while a single `mint`/`burn` pair costs $> 100{,}000$ gas ($\sim$0.003 USD at 30 gwei) — unprofitable by a factor of $> 10^{12}$.

Verdict: theoretically present, economically *infeasible*.

IIb. 8.3 Gas Griefing on Read Path

The log-space `totalOf` adds $\sim$1,100 gas (one `exp()` call) compared to the multiplicative form. An attacker who can force another user's `totalOf` to execute on-chain — e.g., during liquidation — imposes this additional cost on the caller.

Bound: the $+1{,}100$ gas overhead is $< 0.03\%$ of a typical liquidation transaction ($\sim$4.2M gas, Table 20). Liquidation bots operate on profit margins orders of magnitude larger than this cost. The overhead is not controllable by the attacker (it is a fixed function of the `exp()` implementation) and cannot be amplified.

Verdict: measurable but *not exploitable*.

IIb. 8.4 Timestamp Sensitivity and Summary

The yield computation $r_{\text{wad}} = r_{\text{annual}} \times \Delta t/\text{YEAR}$ depends on `block.timestamp`. This is identical in both representations — the log-space form does not change the yield calculation, only how the result is stored. Block proposers can manipulate timestamps by $\sim$12 seconds (one slot), affecting Δt by $< 4 \times 10^{-7}$ of the annual rate per accrual. This bound is *unchanged* from the multiplicative form.

Overall, the log-space transformation has a strictly smaller attack surface. The compounded truncation manipulation vector is eliminated, and no new economically viable attack vector is introduced.

IIb. 9 Limitations and Future Work

Limitations.

1. **Single-protocol validation.** All benchmarks and precision measurements are from XPower Banq. Results may differ for protocols with different accrual frequencies, utilization patterns, or index structures (e.g., Aave's rebasing aTokens, where `balanceOf` is the interest-inclusive view).
2. **Read-path cost increase.** Protocols with on-chain read/write ratios $R > 1.09$ may see a net gas increase (Section IIb. 6.1).
3. **Optimizer-off benchmarks.** Gas measurements were taken with the Solidity optimizer disabled to ensure reproducibility. Optimizer-on results may narrow or widen deltas depending on how `exp()` and `mulDiv` are inlined and optimized.
4. **Self-referential citations.** References [1] and [3] are unpublished XPower Banq preprints. Independent reproduction requires access to the protocol's source code.

IIb. 9.1 Future Work

1. **Cross-protocol benchmarks.** Apply the transformation to a fork of Aave v3 or Compound v3 and measure gas and precision under their accrual patterns and read frequencies.
2. **Optimizer-on gas profile.** Re-run benchmarks with the Solidity optimizer enabled at standard settings (200 runs) to provide production-representative figures.
3. **Formal verification.** A machine-checked proof (e.g., in Certora or Halmos) of the conservation invariant (Property IIb. 4.2) would strengthen confidence for high-TVL deployments.
4. **`ln(1+x)` variant.** For protocols using linear approximation $(1 + r \cdot \Delta t)$ rather than $\exp(r \cdot \Delta t)$, a log-space variant storing $\sum \ln(1 + r_i \cdot \Delta t_i)$ may be preferable. Its gas and precision tradeoffs remain uncharacterised.

IIb. 10 Conclusion

We have presented a log-space compounding index for DeFi lending with five properties:

1. **Overflow elimination**: linear accumulation provides $\sim 5 \times 10^5$ years of headroom at 1000%/yr in the shipped

80-bit field, $\sim 10^{58}$ years in the full `uint256`, vs. $\sim$29 years for the multiplicative form.

2. **Write-path gas saving**: replacing `exp()` with addition saves $\sim$1,200 gas per accrual event ($-114{,}447$ gas net across 23 benchmarked operations).
3. **Precision and security improvement**: zero rounding during accrual; $\leq$ 2 ULP at read-time vs. up to $2N$ ULP of compounded truncation in the multiplicative form. Empirically $+10$–20 wei closer to analytical values. This also eliminates the compounded truncation manipulation vector, yielding a strictly smaller attack surface (Section IIb. 8).
4. **Modular arithmetic compatibility**: the additive structure supports `unchecked` Solidity arithmetic, matching the Uniswap v3 accumulator pattern (Remark IIb. 4.1).
5. **Economic invariant preservation**: all user-visible behaviour — balance queries, interest accrual, lock bonus/malus, liquidation — is mathematically identical to the multiplicative form.

The transformation is algebraically trivial — it applies the identity $\log(\prod \exp(r_i)) = \sum r_i$ — yet its practical consequences are significant. The log-space index is not merely overflow-safe and gas-efficient; it is the *numerically correct* representation for compounding interest in fixed-point arithmetic. The mechanism is deployed in the XPower Banq protocol [1].

XPower Banq: Part III

Mathematical Theory & Proofs

Foundations, Formal Proofs, and Nash Equilibrium Analysis for the XPower Banq Lending Protocol

Abstract

We collect the theoretical foundations of the XPower Banq lending protocol [1]: continuous-compounding interest accrual, log-space TWAP construction, time-weighted parameter integration, token-bucket rate limiting, and formal proofs of cascade attenuation, Sybil rate-limiting, governance rate bounds, and MEV front-running resistance. We then analyse the Nash equilibrium of lock adoption under utilization-dependent rates and a secondary-market discount, characterising two self-reinforcing equilibria (low and high adoption) and proving that the protocol margin remains within sustainable bounds for any aggregate lock fraction $\bar{\rho} \in [0, 1]$. Numerical evaluation of these results is the subject of the companion simulations paper [6].

Keywords: DeFi lending, formal proofs, Nash equilibrium, lock adoption, oracle theory, TWAP

III. 1 Mathematical Foundations

III. 1.1 Interest Accrual

Interest accrues via continuous compounding with ray precision (27 decimals).

Definition III. 1.1 (Position Index)**.** The global index I_t at time t with annual rate r:

$$I_t = I_0 \cdot e^{r \cdot (t - t_0)/T_{\text{year}}} \tag{36}$$

where $T_{\text{year}} = 365.25 \times 86400 = 31{,}557{,}600$ seconds.[1]

Theorem III. 1.1 (Balance Accrual)**.** *A user's total balance B_t given principal P and checkpoint index I_u:*

$$B_t = P \cdot \frac{I_t}{I_u} \tag{37}$$

Remark III. 1.1 (Piecewise integration in practice). In the shipped contracts, r is not a single constant but a function of utilization $r(t) = \hat{r}(\text{util}(t))$, and the index is integrated piecewise:

$$\log I_t - \log I_0 = \sum_i \hat{r}(\text{util}_i) \cdot (t_{i+1} - t_i)/T_{\text{year}}$$

with re-indexings driven by every state-changing call. See `Position._indexOf` and `Position._reindex`; this matches the time-weighted framing of §III. 1.2.

III. 1.2 Time-Weighted Integration

Definition III. 1.2 (Time-Weighted Mean)**.** For parameter values θ_i at times t_i:

$$\bar{\theta}(t) = \frac{\sum_{i=0}^{n-1} \theta_i \cdot (t_{i+1} - t_i) + \theta_n \cdot (t - t_n)}{t - t_0} \tag{38}$$

III. 1.3 Oracle Price Aggregation

Definition III. 1.3 (Log-Space TWAP Quote)**.** Each refresh queries the AMM bidirectionally and computes:

$$m_t = \log_2(\bar{p}_t) \quad \text{(log mid price)} \tag{39}$$
$$s^{AB}_{t,log} = \log_2(1 + s^{AB}_t) \quad \text{(log spread AB)} \tag{40}$$
$$s^{BA}_{t,log} = \log_2(1 + s^{BA}_t) \quad \text{(log spread BA)} \tag{41}$$
$$r_t = \tfrac{1}{2}\left(s^{AB}_{t,log} + s^{BA}_{t,log}\right) \quad \text{(log geomean spread)} \tag{42}$$

The EMA smooths in log space with decay α, blending the previous mean with the *previously stored* sample (the contract maintains a one-sample buffer: `TWAPLib.update` reads `last_mid = midOf(last)` *before* overwriting `twap_.last` with the new m_t, so the new sample enters the mean only at the next refresh):

$$\overline{m}_t = \alpha \cdot \overline{m}_{t-1} + (1{-}\alpha) \cdot m_{t-1} \tag{43}$$
$$\overline{r}_t = \alpha \cdot \overline{r}_{t-1} + (1{-}\alpha) \cdot r_{t-1} \tag{44}$$

where m_{t-1} (resp. r_{t-1}) denotes the most recent sample observed strictly *before* the current refresh.

[1] YEAR = CENTURY/100 with CENTURY = `365_25` days, i.e. $36525 \cdot 86400/100 = 31{,}557{,}600$ s, to keep the constant integer-valued in Solidity.

Definition III. 1.4 (Spread Scaling)**.** Let n = `amount/unit_source` be the position expressed in *whole units* of the source asset (so a 1.5 ETH order has $n = 1.5$, not $1.5 \cdot 10^{18}$). With base spread $s = 2^{\overline{r}} - 1$:

$$x = n \cdot s, \quad \mu = \log_2(2x + 2), \quad s' = s \cdot \mu \tag{45}$$

Final quotes: $\text{bid}(n) = \text{value} - s' \cdot \text{value}/2$, $\text{ask}(n) = \text{value} + s' \cdot \text{value}/2$. Implemented in `Oracle._hlfSpread` / `Oracle._center`.

III. 1.4 Rate Limiting

Definition III. 1.5 (Token Bucket)**.** A rate limiter with capacity $C_{\max}$, regeneration rate 1 unit per wall-clock second (a hard-coded design choice, not a governable parameter), and per-call cost c supplied by the caller (the `floor_cost` argument of each modifier invocation):

$$C'(t) = \min\left(C_{\max}, C(t_{\text{last}}) + (t - t_{\text{last}})\right) - c \tag{46}$$

Operation allowed iff $C'(t) \geq 0$. Implementation: `RateLimited._ratelimitedCheck` regenerates first, reverts if $C(t_{\text{last}}) + (t - t_{\text{last}}) < c$, then commits the post-decrement state—algebraically equivalent to the post-cost residual $C'(t) \geq 0$.

III. 2 Formal Proofs

III. 2.1 Cascade Attenuation Proof

Proof of the Cascade Attenuation Theorem (§4.1 of [1]). Let liquidation of position P with lock fraction ϕ transfer supply tokens S to liquidator L.

Step 1: By `Position.burn`, `redeem` reverts when the post-burn balance would fall below the user's aggregate lock total: the post-condition asserts `totalOf(user)` $\geq$ `_lock.totalOf(user)` after burning the requested amount, raising `Locked(user, total)` otherwise. There is no per-token "locked vs. unlocked" tag; the lock is a reservation against the aggregate balance. Hence the locked share $\phi \cdot S$ of any position cannot leave the protocol; only the unlocked surplus $(1{-}\phi) \cdot S$ is available for redemption and resale.

Step 2: Liquidator L receives position tokens, not underlying assets. The underlying assets remain in the Vault.

Step 3: Only unlocked fraction $(1{-}\phi)$ can generate sell pressure. Price impact: $\Delta p = f((1{-}\phi) \cdot V_{\text{sold}})$.

Step 4: Lock property is preserved through transfer: by `Position._lockPush` → `LockLib.push`, each ring-buffer slot's value is scaled by `mulDiv(src_value, amount, balance)` and added to the target. This is invoked unconditionally on every `transferFrom`, including the liquidation path. Hence the per-user lock fraction ϕ_{user} = `depthOf(user)/balance(user)` is preserved across any transfer of size $\leq$ `balance`. Therefore the cascade amplification factor is bounded by $(1{-}\phi)$. □

III. 2.2 Sybil Rate-Limiting

Definition III. 2.1 (Sybil Resistance Scope)**.** The cap system bounds accumulation *rate* via holder-count scaling.

It does **not** guarantee that more accounts hurts an attacker's long-term equilibrium share—simulation shows <2% penalty over 60 weeks.

Theorem III. 2.1 (Sybil Attack Cost)**.** *Assume each of the k Sybil accounts holds a positive balance of at least one whole position token (`balance` $\geq$ `unit_token` = $10^{decimals}$), so that it is counted as a* large holder *by `Position._largeHolders` (the running counter incremented in `_update` only when an account crosses `balance` $\geq$ `unit`). Assume further that the per-account cap considered is for a* subsequent *deposit on top of an existing positive balance, so that the $\beta(\lambda) = 12\lambda(1-\lambda)^2$ weighting of `_capOf` (gated on `balance > 0 && total > balance`) applies. Then for the attacker to increase total cap gain by factor F, they must create $k > F^2 \cdot (n+2) - (n+2)$ such large-holder Sybils. This quadratic scaling makes large gains expensive.*

Proof. Under the large-holder hypothesis the denominator $\sqrt{\texttt{largeHolders()} + 2}$ scales from $\sqrt{n+2}$ to $\sqrt{n+k+2}$ when k funded Sybils are added; a Sybil account with `balance < unit` would not advance the counter and is excluded by hypothesis. Under the existing-holder hypothesis, the beta block of `_capOf` is active. With 1 account: $C_1 = C_{\max} \cdot \beta(\lambda)/\sqrt{n+2}$. With k equal accounts and small λ/k: $C_k \approx 12\lambda C_{\max}/\sqrt{n+k+2}$. For $C_k > F \cdot C_1$: $\sqrt{n+2}/\sqrt{n+k+2} > F/k$, yielding the stated bound. (A floor `MIN_HOLDERS_ID` on `largeHolders()` prevents the denominator from collapsing when n is small.) □

III. 2.3 Governance Rate Bound Proof

Proof of the Governance Rate Bound (§6 of [1]).
Step 1 (Single bound): Each transition satisfies $\theta_0/\mu \leq \theta_1 \leq \mu \cdot \theta_0$, enforced inline by `Parameterized._setTargetIf`, which compares the proposed value against `old_value << 1` (upper bound, $\mu = 2$) and `old_value >> 1` (lower bound, $1/\mu$) and reverts on violation.

Step 2 (Rate limiting): `setTarget` is guarded by the `limited(Constant.MONTH, ...)` modifier, which restricts updates to once per $\Delta t_{\min}$ = `Constant.MONTH` = `Constant.YEAR/12` $\approx$ 30.4375 days (a compile-time constant, not a governable parameter). In interval $(t-t_0)$, at most $n = \lfloor (t-t_0)/\Delta t_{\min} \rfloor$ transitions occur.

Step 3 (Composition): After n transitions: $\theta_n \leq \mu^n \theta_0$ and $\theta_n \geq \mu^{-n}\theta_0$.

Step 4 (Transitions): During transitions, the time-weighted mean stays between θ_{old} and θ_{new}, preserving the bound. □

Corollary III. 2.2 (Attack Detection Window)**.** *To change a parameter by factor $F > \mu$: $T_{attack} \geq \lceil \log_\mu F \rceil \cdot \Delta t_{\min}$. With $\mu = 2$ and the compile-time constant $\Delta t_{\min}$ = `Constant.MONTH` $\approx$ 30.4375 days (not itself governable), a $10\times$ change requires 4 months.*

III. 2.4 MEV Front-Running Resistance

Theorem III. 2.3 (MEV Front-Running Cost)**.** *A liquidation with profit π cannot be profitably front-run if: $H_{attacker}/H_{victim} < t_{cache}/T_{PoW}(\pi)$ where t_{cache} is the `PowLimited` cache window over which the PoW key is stable.*

Proof. The PoW hash includes `tx.origin`, preventing solution reuse across attacker EOAs. The `PowLimited` modifier caches the block hash anchoring the PoW key for `cacheTime` seconds, refreshing only once `cacheTime + _blockTime < block.timestamp`. Both `Pool` and `Oracle` instantiate `PowLimited(1 hours)`, so t_{cache} = 1 hour = 3600 s. With expected solve time $T_{\text{PoW}}(\pi) = 16^d/H$, the attacker must mine independently within t_{cache} against its own `tx.origin`; front-running fails when $T_{\text{PoW}} > t_{\text{cache}}$, i.e. when difficulty d is tuned so that $16^d/H_{\text{attacker}} > t_{\text{cache}} = 3600$ s. □

III. 3 Nash Equilibrium Analysis for Lock Adoption

This section analyzes the game-theoretic equilibrium of lock adoption. We model rational actors choosing between locked and unlocked positions, accounting for APY differentials, secondary market dynamics, and liquidation seniority.

III. 3.1 Model Setup

III. 3.1.1 Position Types and Rates

With spread $s = 10\%$ and $r_{\text{bonus}} = r_{\text{malus}} = s$:

Position Type	Rate Multiplier	APY vs. Base
Unlocked Supplier	$(1-s) = 0.90$	−10%
Locked Supplier	$(1+r_{\text{bonus}})(1-s) = 0.99$	−1%
Unlocked Borrower	$(1+s) = 1.10$	+10%
Locked Borrower	$(1-r_{\text{malus}})(1+s) = 0.99$	−1%

The **APY differential** between locked and unlocked positions:

$$\Delta r = r_{\text{base}} \times (1-s) \times r_{\text{bonus}} = r_{\text{base}} \times 0.9 \times 0.1 = 0.09 \times r_{\text{base}} \quad (47)$$

III. 3.1.2 Secondary Market Dynamics

Position tokens are ERC20-transferable, enabling secondary market trading. However, locked positions trade at a **discount** D relative to NAV because:

1. Locked positions cannot be redeemed for underlying assets
2. Exit is only possible via secondary market sale
3. Lock status transfers proportionally with tokens (buyers receive locked tokens)

The discount D is endogenous to market conditions, typically ranging from 3–10%[2] based on:

[2] Empirical range observed in secondary markets for illiquid DeFi positions: Lido stETH traded at 6–7% discount during the June 2022 liquidity crisis (Nansen, "On-Chain Forensics: Demystifying stETH's De-peg," 2022); Convex cvxCRV has traded at 5–12% discount to CRV due to permanent lock-up (geeogi, "cvxCRV peg," 2022).

- Secondary market liquidity depth
- Expected holding period of market participants
- Overall lock adoption rate (affecting liquidity)

III. 3.2 Player Utility Functions

III. 3.2.1 Supplier Utility

For a supplier with lock ratio $\rho \in [0, 1]$ and holding period T:

Unlocked position ($\rho = 0$):

$$U_{\text{unlocked}} = P \times (1 + r_{\text{unlocked}})^T \tag{48}$$

Locked position ($\rho = 1$):

$$U_{\text{locked}} = P \times (1 + r_{\text{locked}})^T \times (1 - D) \tag{49}$$

where the $(1 - D)$ factor reflects the secondary market discount on exit.

III. 3.2.2 Breakeven Condition

Lock adoption is rational when $U_{\text{locked}} > U_{\text{unlocked}}$:

$$(1 + r_{\text{locked}})^T \times (1 - D) > (1 + r_{\text{unlocked}})^T \tag{50}$$

Solving for the **breakeven holding period** T^*:

$$T^* = \frac{\ln(1 - D)}{\ln(1 + r_{\text{unlocked}}) - \ln(1 + r_{\text{locked}})} \approx \frac{D}{\Delta r} \tag{51}$$

III. 3.3 Breakeven Analysis by Utilization

The base interest rate r_{base} varies with utilization U according to the piecewise-linear model (§4.7 of [1]). Table 23 presents breakeven periods for various utilization levels.

Table 23: Breakeven holding periods by utilization

Utilization U	Base Rate r	Differential Δr	T^* $(D = 5\%)$	T^* $(D = 3\%)$
80%	8.9%	0.80%	6.3y	3.8y
90%	10%	0.90%	5.6y	3.3y
95%	55%	4.95%	12m	7m
98%	82%	7.38%	8m	5m
100%	100%	9.0%	6.7m	4m

Lock adoption thus becomes economically attractive primarily during high-utilization periods ($U > 95\%$), precisely when cascade prevention is most valuable.

III. 3.4 Liquidation Seniority Value

Locked positions gain *de facto* seniority in liquidations because liquidators prefer unlocked positions (immediate liquidity). The seniority value $S(\bar{\rho})$ depends on aggregate lock adoption $\bar{\rho}$ (denoted ϕ in the Cascade Attenuation Theorem (§4.1 of [1])):

$$S(\bar{\rho}) \approx P(\text{liquidation}) \times (1 - \bar{\rho}) \times L_{\text{loss}} \tag{52}$$

where $P(\text{liquidation})$ is the probability of a liquidation event and L_{loss} is the expected loss in liquidation.

This creates a **coordination game**: seniority value decreases as lock adoption increases. At $\bar{\rho} = 0$, locked positions have maximum seniority; at $\bar{\rho} = 1$, no differentiation exists.

III. 3.5 Nash Equilibrium Characterization

III. 3.5.1 Equilibrium Condition

At equilibrium lock adoption $\bar{\rho}^*$, the marginal user is indifferent between locking and not locking:

$$\underbrace{\Delta r \times T}_{\text{APY benefit}} + \underbrace{S(\bar{\rho}^*)}_{\text{Seniority}} = \underbrace{D}_{\text{Discount cost}} \tag{53}$$

III. 3.5.2 Utilization-Dependent Equilibria

Table 24: Equilibrium lock adoption by utilization regime

Utilization Regime	Base Rate	Expected $\bar{\rho}^*$	Protocol Margin
Normal ($U \in [0, 90\%)$)	$< 10\%$	10–20%	16–18% of r
Elevated ($U \in [90\%, 95\%)$)	10–55%	20–40%	12–16% of r
Stressed ($U \in [95\%, 100\%]$)	$> 55\%$	40–70%	6–12% of r

The mechanism thus aligns incentives with protocol needs: under normal conditions, low lock adoption preserves revenue, while under stressed conditions, high adoption provides cascade protection when it is most needed.

III. 3.6 Protocol Margin Analysis

III. 3.6.1 Margin as Function of Lock Adoption

Protocol margin per unit interest flow:

$$M(\bar{\rho}) = 2s - r_{\text{bonus}} \times \bar{\rho}^S - r_{\text{malus}} \times \bar{\rho}^B \tag{54}$$

With $r_{\text{bonus}} = r_{\text{malus}} = s$ and assuming $\bar{\rho}^S = \bar{\rho}^B = \bar{\rho}$:

$$M(\bar{\rho}) = 2s(1 - \bar{\rho}) \tag{55}$$

Table 25: Protocol margin by lock adoption

Lock Adoption $\bar{\rho}$	Margin	Retention
0%	20% of r	100%
25%	15% of r	75%
50%	10% of r	50%
75%	5% of r	25%
100%	0% of r	0%

III. 3.6.2 Solvency at Full Adoption

The default configuration operates at the solvency boundary:

$$r_{\text{bonus}} + r_{\text{malus}} = s + s = 2s \quad \checkmark \tag{56}$$

Protocol solvency is maintained for any $\bar{\rho} \in [0,1]$, though margin approaches zero as $\bar{\rho} \to 1$.

III. 3.7 Stability Analysis

III. 3.7.1 Multiple Equilibria

The lock adoption game exhibits **multiple equilibria** due to the endogenous discount:

1. **Low-adoption equilibrium** ($\bar{\rho}^* \approx 10\%$): Thin secondary market $\Rightarrow$ high discount D $\Rightarrow$ locking unattractive $\Rightarrow$ low adoption (self-reinforcing)
2. **High-adoption equilibrium** ($\bar{\rho}^* \approx 60\%$): Deep secondary market $\Rightarrow$ low discount D $\Rightarrow$ locking attractive $\Rightarrow$ high adoption (self-reinforcing)

III. 3.7.2 Equilibrium Selection

The realized equilibrium depends on:

- **Initial conditions**: Early lock adoption seeds secondary market liquidity
- **Utilization dynamics**: High-utilization periods push toward high-adoption equilibrium
- **External liquidity**: Protocol-seeded liquidity can bootstrap the high-adoption equilibrium

III. 3.8 Summary

The lock mechanism creates a self-regulating system in which rational actors provide cascade protection precisely when the protocol most needs it.

Theorem III. 3.1 (Lock Adoption Equilibrium). *With $r_{bonus} = r_{malus} = s = 10\%$:*

1. *Lock adoption is **utilization-dependent**: attractive when $U > 95\%$, marginal otherwise*
2. *The APY differential $\Delta r = 9\% \times r_{base}$ provides maximum incentive within solvency constraints*
3. *Lock adoption **self-regulates**: increases during stress (cascade protection) and decreases during calm (revenue preservation)*
4. *Protocol margin ranges from 20% of r (no locks) to 0% of r (full locks), with expected operating range 8–16% of r*

The secondary market discount acts as a natural governor—preventing full adoption while ensuring sufficient incentive for meaningful cascade protection—so the mechanism achieves incentive alignment without external intervention.

The welfare balance is favorable. Cascade attenuation reduces depth by factor $(1-\phi)$, and spread compression rewards committed participants—at full lock with $s = r_{\text{bonus}} = r_{\text{malus}} = 10\%$, both locked suppliers and borrowers face effective rate multiplier 0.99, approaching a zero-spread market. Against these benefits stands liquidity fragmentation: locked supply raises effective utilization to $U_{\text{eff}} = V_b/(V_s(1-\phi))$, potentially triggering kink-rate dynamics at lower borrow volumes. The interest rate curve absorbs this naturally—higher effective utilization raises rates, incentivizing settlement—so the effect narrows the sub-kink operating range without creating instability.

Together, these properties confirm Theorem III. 3.1: the mechanism is self-regulating, with per-parameter solvency bounds that require no on-chain adoption tracking, cascade dynamics that naturally throttle liquidation velocity, and welfare effects that the interest rate curve absorbs without instability.

III. 3.8.1 Equilibrium Structure

The game exhibits two self-reinforcing equilibria: high adoption ($\bar{\rho} > 40\%$, margin 8–12% of r, strong cascade protection) and low adoption ($\bar{\rho} < 15\%$, margin 16–20% of r, high cascade exposure). Selection is path-dependent—seeding secondary market liquidity bootstraps the high-adoption regime, while neglect traps the protocol in low adoption. Governance can influence this by initially setting r_{bonus} above the long-run target, then reducing it via lethargic transition once adoption stabilizes.

Within either regime, the mechanism is countercyclical: lock adoption rises above $U = 95\%$ when cascade prevention is most valuable, then recedes as utilization normalizes, restoring margin. Lethargic governance reinforces this—the 0.5x–2x per-cycle parameter bound ensures that equilibrium shifts occur gradually enough for the secondary market to adjust. During stress, protocol margin decreases as the intended trade-off for cascade protection, and recovers during the subsequent normalization.

The equilibrium is asymmetric across position types. Locked suppliers earn at effective rate $r_{\text{base}}(1-s)(1+r_{\text{bonus}} \cdot \rho_u)$ but must compensate the secondary market discount D, requiring $U > 95\%$ for rational adoption. Locked borrowers pay at effective rate $r_{\text{base}}(1+s)(1-r_{\text{malus}} \cdot \rho_u)$—reducing an *obligation* with no redemption friction—making $\bar{\rho}^B > \bar{\rho}^S$ in the normal regime, converging during stress as the supply bonus dominates. A plausible operating point ($\bar{\rho}^S \approx 15\%$, $\bar{\rho}^B \approx 35\%$) yields:

$$M = 2s - s \cdot \bar{\rho}^S - s \cdot \bar{\rho}^B = 20\% - 1.5\% - 3.5\% = 15\% \text{ of } r \tag{57}$$

within the sustainable range of Table 25.

III. 3.8.2 Cascade Dynamics and Keeper Sizing

Debt assumption transfers locks proportionally, preserving aggregate $\bar{\rho}$ through cascades—the attenuation bound $(1-\phi)$ from Theorem 4.1 of [1] holds dynamically, not merely at a pre-crash snapshot. At $\bar{\rho} \approx 30\%$, this reduces 25%-shock liquidations from 85.7% to ≈55% of positions (Table 5 of [1]). However, locked supply received by a liquidator cannot be redeemed: absorbing a fully-locked victim at exponent e increases assumed debt by $d = V_b \cdot 2^{-e}$

while adding zero redeemable supply, eroding headroom for subsequent liquidations.

The exponent choice is itself strategic. Choosing $e = 0$ captures the full position—maximizing per-event profit but inheriting the full lock fraction. Choosing $e \geq 1$ halves the position taken, reducing lock inheritance at the cost of leaving the remainder available to competitors. In equilibrium, the optimal e reflects each liquidator's current health factor and the remaining cascade depth: unconstrained liquidators with ample headroom prefer small e, while capacity-constrained ones ration across targets. Both effects—lock removal from the liquidation pool and liquidator self-throttling—reinforce cascade attenuation.

Keeper sizing must account for lock propagation. A keeper processing k sequential liquidations needs headroom for both aggregate debt absorbed and the locked fraction that cannot be redeemed. If the expected victim lock fraction is $\bar{\rho}$, the effective headroom requirement per liquidation increases by factor $1/(1-\bar{\rho})$ relative to the unlocked case, since only $(1-\bar{\rho})$ of received supply contributes redeemable collateral. The cascade simulation in the companion simulations paper [6] quantifies this across varying lock fractions and crash severities.

III. 3.8.3 Solvency Guarantees

Protocol solvency is enforced algebraically. The supervised contract enforces $r_{\text{bonus}} \leq s$ and $r_{\text{malus}} \leq s$ in *both* directions via `_setTarget` overrides: when setting `LOCK_BONUS_ID`/`LOCK_MALUS_ID`, the new value must be $\leq s$; when setting `SPREAD_ID`, the new spread must remain $\geq \max(r_{\text{bonus}}, r_{\text{malus}})$, so a governance-driven *decrease* of s cannot violate the invariant either. Additionally $s \leq$ `Constant.HLF` $= 0.5$ is enforced. Since $\bar{\rho}^S, \bar{\rho}^B \in [0, 1]$, the aggregate condition $r_{\text{bonus}}\bar{\rho}^S + r_{\text{malus}}\bar{\rho}^B \leq 2s$ holds without on-chain lock-adoption tracking. The worst case ($\bar{\rho}^S = \bar{\rho}^B = 1$) yields zero margin but never insolvency—avoiding the gas overhead of global lock-adoption statistics by relying on per-parameter bounds enforced at governance time. Lock adoption also tightens beta-distributed caps indirectly: locked suppliers are persistent holders, inflating the effective n in the $\sqrt{n+2}$ divisor, while attackers splitting across k accounts face the $O(\sqrt{k})$ cap-scaling barrier *and* irrevocability—acquiring locked positions via secondary market incurs discount D, creating a multiplicative barrier to rapid concentration.

III. 3.8.4 Welfare

The welfare balance is favorable. Cascade attenuation reduces depth by factor $(1-\phi)$, and spread compression rewards committed participants—at full lock with $s = r_{\text{bonus}} = r_{\text{malus}} = 10\%$, both locked suppliers and borrowers face effective rate multiplier 0.99, approaching a zero-spread market. Against these benefits stands liquidity fragmentation: locked supply raises effective utilization to $U_{\text{eff}} = V_b/(V_s(1-\phi))$, potentially triggering kink-rate dynamics at lower borrow volumes. The interest rate curve absorbs this naturally—higher effective utilization raises rates, incentivizing settlement—so the effect narrows the sub-kink operating range without creating instability.

Together, these properties confirm Theorem III. 3.1: the mechanism is self-regulating, with per-parameter solvency bounds that require no on-chain adoption tracking, cascade dynamics that naturally throttle liquidation velocity, and welfare effects that the interest rate curve absorbs without instability.

XPower Banq: Part IV

Simulations & Risk Analysis

Capacity Accumulation, Cascade, TWAP, and Bad-Debt Simulations for the XPower Banq Lending Protocol

Abstract

We present the simulation suite that empirically validates the XPower Banq lending protocol [1]. Four studies are reported: (1) capacity accumulation under the beta-distributed cap function, confirming $O(\sqrt{n})$ Sybil-resistance scaling; (2) liquidation-cascade simulations across five market-impact regimes, demonstrating up to 80% cascade reduction at full lock adoption; (3) log-space TWAP oracle response to price shocks and liquidity changes, including 60 on-chain Foundry scenario tests showing two-tick manipulation immunity; and (4) bad-debt risk quantification via Merton jump-diffusion Monte Carlo, with closed-form analytical bound and a five-configuration safe operating region. The mathematical foundations underlying these simulations are developed in the companion theory paper [5].

Keywords: DeFi simulation, Monte Carlo, jump diffusion, TWAP, cascade dynamics, bad debt, Sybil resistance

IV. 1 Capacity Accumulation Simulation

The following Python script simulates capacity accumulation under the beta-distributed cap function (the beta cap formula (§4.4 of [1])).

```
#!/usr/bin/env python3
from numpy import sqrt

def my_cap(balance, max_cap, supply, n_accounts) -> float:
    """
    Computes the max. individual capacity gain per round
    for given balance, supply, max. capacity and accounts.
    """
    # compute lambda: balance / supply
    lmb = balance / supply
    # compute my capacity gain multiplier
    mul = 12*lmb*(1-lmb)**2
    # compute my capacity gain divisor
    div = sqrt(n_accounts + 2)
    # compute my capacity gain
    return max_cap * mul / div

if __name__ == "__main__":
    # epsilon for stopping criterion
    eps = 1e-3
    # max. capacity (normalized)
    max_cap = 1.00
    # supply (normalized)
    supply = 1.00

    print(f"beg_balance;n_accounts;iterations")
    for beg_balance in map(lambda e: 10**(-e), range(1,9)):
        for n_accounts in map(lambda e: 10**(2*e), range(1,5)):

            end_balance = beg_balance
            iterations = 0

            # simulate cap gain until close to supply
            while (supply - end_balance) / supply > eps:
                end_balance += my_cap(
                    end_balance, max_cap, supply, n_accounts
                )
                iterations += 1

            print(f"{beg_balance:.0e};{n_accounts:.0e};{
                iterations:.3e}")
```

Listing 12: Capacity accumulation simulation

IV. 1.1 Cap Function Components

The `my_cap` function implements the beta cap formula (§4.4 of [1]). The balance ratio $\lambda = B/S$ normalizes across token amounts and decimals. The Beta(2,3) multiplier $12\lambda(1-\lambda)^2$ vanishes at both boundaries ($\lambda = 0$ and $\lambda = 1$), peaks at $\lambda = 1/3$, and decays quadratically via $(1-\lambda)^2$ as holdings approach monopoly. The holder divisor $\sqrt{n+2}$ provides sublinear Sybil resistance: creating more accounts increases n, reducing per-account cap gains, with the offset of 2 preventing degeneracy at low holder counts. $C_{\max}$ is the governance-configured upper bound on relative capacity. In the protocol, max_cap is the residual capacity `_relLimit(abs_limit, total)`; the simulation's normalised `max_cap = 1.00, supply = 1.00` corresponds to the limiting case where `abs_limit` $\gg$ `total` so the headroom is approximately `abs_limit`.

IV. 1.2 Simulation Algorithm

The simulation sweeps over starting balances and holder counts. For each combination, it iteratively adds the cap gain (via `my_cap`) to the user's balance until the remaining capacity $(1 - \lambda)$ falls below $\varepsilon = 0.1\%$ of total supply, recording the number of iterations required. The stopping threshold avoids infinite loops caused by the asymptotic decay of the cap function as $\lambda \to 1$. The holder count n is held constant throughout each run; in practice, n would fluctuate as users enter and exit.

IV. 1.3 Convergence Behavior

Near $\lambda = 0$, the cap gain grows linearly ($\approx 12\lambda/\sqrt{n+2}$), producing a fast start. Near $\lambda = 1$, it shrinks quadratically ($\approx 12(1-\lambda)^2/\sqrt{n+2}$), producing a slow approach to full capacity. Because most iterations occur in this slow-convergence region, starting balance has minimal impact on total iterations. Since cap gain scales as $1/\sqrt{n}$, iterations scale as $\sqrt{n}$—explaining the $10\times$ increase when moving from 10^2 to 10^4 accounts.

IV. 1.4 Representative Output

Table 26: Simulation results: iterations to full capacity

Starting λ	Accounts (n)	Iterations
10^{-2}	10^2	846
10^{-2}	10^4	8,417
10^{-2}	10^6	84,200
10^{-2}	10^8	842,100
10^{-4}	10^2	852
10^{-4}	10^4	8,458
10^{-4}	10^6	84,590
10^{-4}	10^8	845,900

The results confirm the $O(\sqrt{n})$ scaling: 10^4 accounts require approximately $10\times$ more iterations than 10^2 accounts. Starting balance has minimal impact (<2% across 7 orders of magnitude), confirming that convergence near $\lambda = 1$ dominates. For a protocol with 10^6 large holders, any user requires approximately 84,000 iterations to reach full capacity, providing predictable growth independent of initial position size.

IV. 1.5 Multi-Account Share Dynamics

The following simulation examines how capital shares evolve when multiple accounts compete for capacity under the beta cap function and iteration constraints:

```
#!/usr/bin/env python3
from numpy import sqrt

def beta_cap(balance, max_cap, supply, n_holders):
    """Beta-distributed cap function (Equation 4)."""
    lam = balance / supply if supply > 0 else 0
    mul = 12 * lam * (1 - lam) ** 2
    div = sqrt(n_holders + 2)  # Per Eq. 4
    return max_cap * mul / div

def simulate_shares(
    n_honest, k_sybil,
    honest_capital, sybil_capital,
    iters_per_week=7, weeks=60, max_cap=1.0
):
    """Simulate share evolution with iteration cap."""
    n_total = n_honest + k_sybil
    honest_bal = [honest_capital / n_honest] * n_honest
    sybil_bal = [sybil_capital / k_sybil] * k_sybil
    total_supply = honest_capital + sybil_capital

```

```
    trajectory = []
    for week in range(weeks):
        for _ in range(iters_per_week):
            honest_bal = [
                b + beta_cap(b, max_cap, total_supply, n_total)
                for b in honest_bal
            ]
            sybil_bal = [
                b + beta_cap(b, max_cap, total_supply, n_total)
                for b in sybil_bal
            ]
            total_supply = sum(honest_bal) + sum(sybil_bal)

        honest_share = sum(honest_bal) / total_supply
        sybil_share = sum(sybil_bal) / total_supply
        trajectory.append((week, honest_share, sybil_share))

    return trajectory
```

Listing 13: Multi-account share dynamics simulation

The simulation reveals that capital shares are *persistent*, not convergent: the beta cap function preserves relative capital ratios because capacity gain is proportional to current $\lambda = B/S$.

Table 27: Simulated share evolution (100 honest vs. 50 Sybil accounts)

Scenario	Initial Share	Week 7	Week 59
Equal shares	50%:50%	50.9%:49.1%	51.4%:48.6%
Uneq. shares	10.5%:89.5%	11.2%:88.8%	11.9%:88.1%

The results show that initial capital distribution strongly influences long-term shares: with a $9\times$ capital advantage, the advantaged party retains 88% share after 60 weeks. Shares change by less than 2% over 420 iterations, and the weekly cap is iteration-count bounded (7 per week), so burst timing is irrelevant. The protocol's Sybil resistance therefore derives from the $\sqrt{n+2}$ divisor and iteration caps rather than from equilibrium convergence—early capital advantages persist, and the defense operates through structural constraints.

IV. 1.6 Limitations

The simulation assumes static holder count (in practice, n fluctuates), continuous cap gains (real protocols have discrete rounding), no holder floor dynamics (the bootstrap phase with $n_{\text{real}} < n_{\text{min}}$ is unmodeled), and homogeneous iteration timing (real-world timing variations may introduce minor perturbations). Specifically, `Position.largeHolders()` returns $\max(\text{actual}, \texttt{MIN_HOLDERS_ID})$; results in Tab. 26 assume $\texttt{MIN_HOLDERS_ID} = 0$ (no floor). With the deployed floor, iteration counts at low n are uniformly larger by the ratio $\sqrt{(\texttt{MIN_HOLDERS_ID} + 2)/(n + 2)}$. The simulation validates two key properties from §4.4 of [1]: (1) the $O(\sqrt{n})$ iteration scaling, and (2) starting balance independence for single-account accumulation time. It also reveals that multi-account share dynamics do not converge to account-proportional equilibrium—a finding that informs the rate-limiting analysis in §4.4 of [1].

IV. 2 Cascade Simulation Implementation

This appendix provides the cascade simulation implementation. Both scripts share a common `simulate` function and output semicolon-separated values.

IV. 2.1 Multi-Scenario Comparison

The following script compares lock effectiveness across five market liquidity scenarios at a fixed 20% price shock:

```
#!/usr/bin/env python3
from numpy import clip, random

def simulate(positions, price_drop, impact_coef):
    """Simulates_liquidation_cascade_with_market_impact."""
    initial_supply = sum(p[0] for p in positions)
    price = 1.0 - price_drop
    total_liquidated = 0
    while True:
        underwater = [p for p in positions
            if p[0] > 0 and (p[0] * price) / p[1] < 1.0]
        if not underwater:
            break
        round_sold = 0
        for p in underwater:
            unlocked = p[0] * (1 - p[2])
            round_sold += unlocked
            total_liquidated += p[0]
            p[0] = 0
            p[1] = 0
        price -= impact_coef * round_sold
        if price <= 0:
            break
    return (total_liquidated / initial_supply * 100
            ) if initial_supply > 0 else 0

def gen_positions(seed, n, lock):
    """Generate_positions_with_given_lock_fraction."""
    random.seed(seed)
    sizes = random.lognormal(mean=0, sigma=1, size=n)
    healths = clip(random.normal(1.5, 0.3, n), 1.0, 3.0)
    return [[s, s/h, lock] for s, h in zip(sizes, healths)]

if __name__ == "__main__":
    n, seed, drop = 1000, 42, 0.20
    scenarios = [("Liquid", 1e-6, 0.2),
                 ("Moderate", 6e-5, 10.1),
                 ("Strong", 1.5e-4, 25.2),
                 ("Severe", 3e-4, 50.5),
                 ("Extreme", 5e-4, 84.1)]
    print("scenario;depth;k;liq0;liq1;reduction")
    for name, k, depth in scenarios:
        liq0 = simulate(gen_positions(seed, n, 0.0), drop, k)
        liq1 = simulate(gen_positions(seed, n, 1.0), drop, k)
        reduction = (liq0-liq1)/liq0 * 100 if liq0 > 0 else 0
        print(f"{name};{depth:.1f};{k:.0e};{liq0:.1f};{liq1:.1f
            };{reduction:.1f}")
```

Listing 14: Multi-scenario cascade simulation

IV. 2.2 Detailed Per-Scenario Analysis

The following script generates detailed results for a specific scenario, varying both lock fraction and price shock:

```
#!/usr/bin/env python3
import sys
from numpy import clip, random

# simulate() and gen_positions() same as above (omitted)

SCENARIOS = [("Liquid", 1e-6, 0.2),
             ("Moderate", 6e-5, 10.1),
             ("Strong", 1.5e-4, 25.2),
             ("Severe", 3e-4, 50.5),
             ("Extreme", 5e-4, 84.1)]

if __name__ == "__main__":
    idx = int(sys.argv[1])  # 0-4
    name, k, depth = SCENARIOS[idx]
```

```
16      n, seed = 1000, 42
17      locks = [0.00, 0.25, 0.50, 0.75, 1.00]
18      drops = [0.10, 0.15, 0.20, 0.25, 0.30]
19      print("lock;drop;liquidated")
20      for lock in locks:
21          for drop in drops:
22              result = simulate(gen_positions(seed, n, lock),
                    drop, k)
23              print(f"{lock:.2f};{drop:.2f};{result:.3f}")
```

Listing 15: Detailed cascade simulation

IV. 2.3 Algorithm

The `simulate` function iterates a liquidation cascade: after applying the initial price drop, it identifies underwater positions ($H < 1$), liquidates them (computing unlocked collateral as supply $\times (1-\phi)$), applies linear market impact $p \leftarrow p - k \cdot V_{\text{sold}}$, and repeats until no underwater positions remain or price reaches zero. Here H denotes the weighted health $H_{\text{ltv}} = (\text{supply} \cdot w_s)/(\text{borrow} \cdot w_b)$ that the contract checks at `Pool.sol:519` (`wnav_supply < wnav_borrow`). The simulation does not separately parametrise (w_s, w_b); results are LTV-agnostic in this normalisation.

IV. 2.4 Market Impact Model

The linear price impact coefficient k determines market depth ($1/k$ in supply units). The five scenarios range from *Liquid* ($k = 10^{-6}$, pool is 0.2% of depth, no cascade amplification) through *Strong* ($k = 1.5 \times 10^{-4}$, 25% of depth, 46% reduction from locks) to *Extreme* ($k = 5 \times 10^{-4}$, 84% of depth, 80% reduction). The locked fraction ϕ attenuates market impact by factor $(1-\phi)$, directly reducing cascade sell pressure.

IV. 2.5 Limitations

The simulation assumes complete (rather than partial 2^{-e}) liquidation, a linear impact model (real markets exhibit convex slippage), greedy liquidators (real behavior may involve front-running or strategic timing), and a single collateral asset (cross-collateral pools may exhibit different dynamics).

IV. 3 TWAP Oracle Simulations

This appendix presents simulation results for the log-space EMA smoothing and logarithmic spread scaling mechanisms. The oracle stores $\log_2(\text{mid})$ and $\log_2(1 + s_{\text{geo}})$, smoothed via EMA in log space. All simulations use 18-decimal precision (1 unit $= 10^{18}$), matching the Solidity implementation.

IV. 3.1 Decay Factor Visualization

The EMA decay factor $\lambda = 0.5^{1/h}$ determines how quickly the TWAP responds to price changes, where h is the half-life in refresh periods. As Figure 17 shows, λ approaches 1.0 asymptotically with increasing half-life. The steep initial rise means that small changes in low h values have large effects on responsiveness, while values above $h = 20$ yield diminishing returns. In practice, λ values between 0.9 and 0.98 provide the useful operating range: below 0.9 the oracle becomes too reactive, while above 0.98 response times may be too slow for timely liquidations.

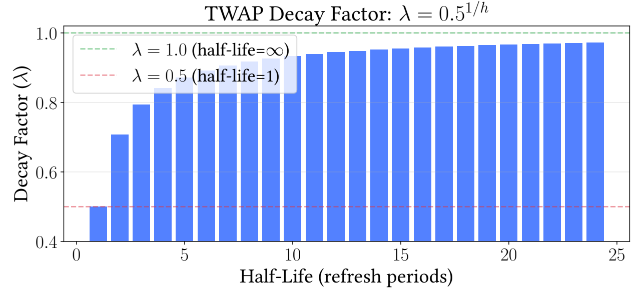


Figure 17: Decay factor λ by half-life. Higher half-life values produce decay factors closer to 1.0, making the TWAP more stable but slower to respond. The protocol default uses $h = 12$ periods ($\text{DECAY_12HL} = 0.5^{1/12} \approx 0.943874$, rounded to $\lambda \approx 0.944$ in subsequent prose).

IV. 3.2 Price Shock Response

Figures 18 and 19 demonstrate how the TWAP mean responds to a sudden price shock (100 → 150) under different half-life configurations.

The half-life h controls the trade-off between responsiveness and manipulation resistance. Shorter half-lives track market movements closely but are more vulnerable to manipulation; longer half-lives filter transient spikes but delay response to genuine shifts. The log-space EMA update $\bar{q}_{n+1} = \lambda \cdot \bar{q}_n + (1-\lambda) \cdot q_n$ ensures smooth convergence without overshooting, with the log transform dampening large deviations more aggressively than a linear EMA. As shown in Figure 18, even with HL=24 the TWAP reaches 90% of the new price within approximately 55 periods.

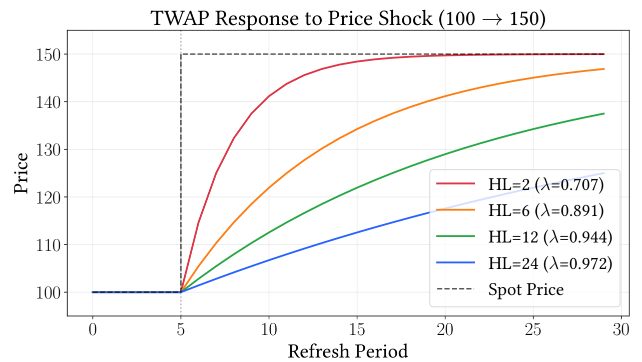


Figure 18: TWAP response to a 50% price shock at period 5. Shorter half-lives (HL=2) converge quickly but offer less manipulation resistance; longer half-lives (HL=24) provide robust smoothing but respond slowly.

The memory decay λ^n quantifies how quickly historical prices lose influence: after k half-lives the residual weight is 0.5^k (12.5% after 3, <1% after 7). For $h = 12$, an attacker must sustain artificial prices for approximately 40 periods to shift the TWAP by 90%.

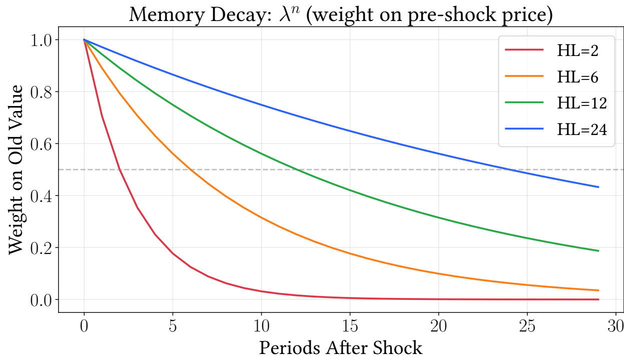


Figure 19: Memory decay showing the weight on pre-shock price over time. After HL periods, the weight on the old value drops to 50%. Higher half-lives retain more memory of historical prices.

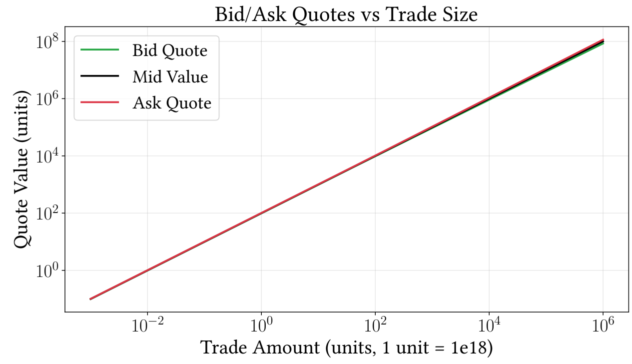


Figure 21: Bid/ask quotes diverge from mid-value as trade size increases. The spread between bid and ask widens logarithmically, protecting against large position manipulation.

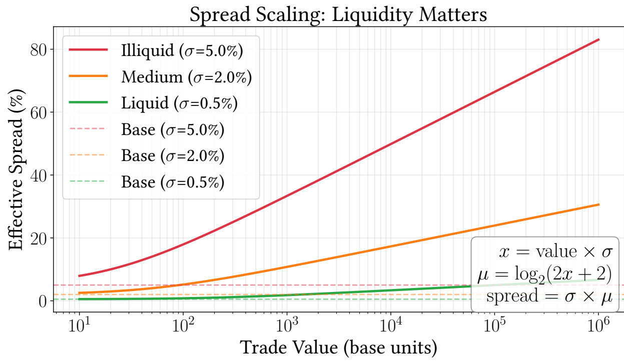


Figure 20: Spread scaling by liquidity level. Illiquid pairs (high base spread) experience faster spread growth with trade size.

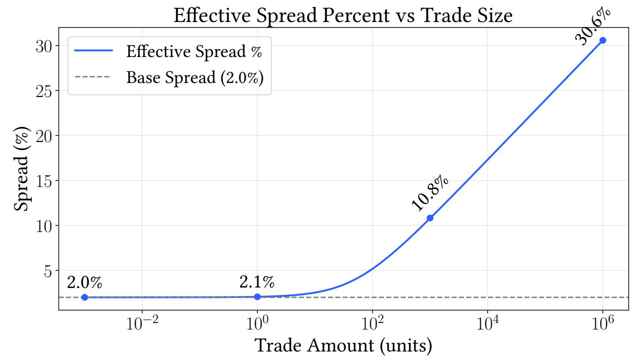


Figure 22: Effective spread grows logarithmically from the base spread (2.0%). At 1 unit the spread is 2.0%; at 1,000 units it reaches 10.8%; at 1,000,000 units it reaches 30.6%.

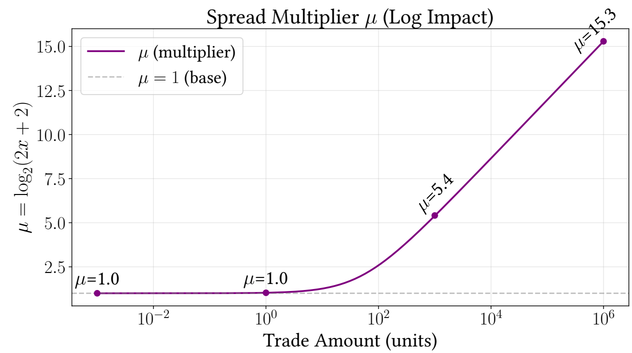


Figure 23: The multiplier $\mu = \log_2(2x+2)$ as a function of $x = n \cdot s$. This function provides bounded logarithmic growth: $\mu = 1$ at $x = 0$, growing to $\mu \approx 21$ at $x = 10^6$.

IV. 3.3 Spread Scaling by Liquidity

The effective spread depends on both position size and market liquidity (captured by the base spread σ). Figure 20 shows how different liquidity levels affect spread scaling.

The base spread σ serves as a proxy for market depth. The formula $\mu = \log_2(2x+2)$ with $x = n \cdot \sigma$ provides self-calibrating behavior: liquid markets (low σ) tolerate larger positions before significant spread widening, eliminating the need for per-pair parameter tuning while ensuring conservative valuations in thin markets.

IV. 3.4 Complete Spread Analysis

Figures 21, 22, and 23 provide a comprehensive view of the spread scaling mechanism using a base spread of 2%.

Together, these three views illustrate that while large positions incur wider spreads, the logarithmic scaling keeps them economically reasonable.

IV. 3.5 On-Chain Scenario Testing

To complement the preceding simulations, we conducted comprehensive on-chain scenario testing of the deployed oracle contract using Foundry. The test suite exercises 10 attack and stress scenarios across 6 token/liquidity configurations (60 total tests), measuring mid-price deviation and spread response under adversarial conditions. All tests use the production DECAY_12HL ($\alpha \approx 0.944$, half-life

h=12 refreshes) with 1-hour refresh intervals. Tables 29 and 30 summarize the results; all values are averages across the 6 configurations unless noted.

IV. 3.5.1 Token/Liquidity Configurations

The test matrix covers symmetric and asymmetric reserves across all supported decimal combinations:

Table 28: Oracle scenario test configurations

Decimals	Reserves	Profile
18/18	100/100	Symmetric standard
18/18	180k/50	Asymmetric standard
6/6	100/100	Symmetric stablecoin
6/6	180k/50	Asymmetric stablecoin
18/6	100/100	Mixed (18→6)
6/18	100/100	Mixed (6→18)

IV. 3.5.2 Mid-Price Manipulation Resistance

Table 29 records the mid-price deviation $\Delta\%$ from baseline after each scenario, measuring how far an attacker (or legitimate market move) can shift the oracle's reported price.

Table 29: Mid-price deviation (%) after each scenario

Scenario	Description	$\Delta\%$
S01	Benign $\pm$5% oscillations, 24 refreshes	2.3%
S02	Instant 2× jump, 14 sustained refreshes	22.5%
S03	Gradual 2× drift over 12 refreshes	9.1%
S04	2× spike, immediate return to baseline	3.7%
S05	2× spike, gradual return over 12 refreshes	14.9%
S06	10× liquidity increase (same ratio)	$<$0.01%
S07	Gradual 10× liquidity increase	$<$0.01%
S08	90% liquidity drain (same ratio)	$<$0.06%
S09	Gradual 90% liquidity drain	$<$0.01%
S10a	3× manipulation between refreshes	**0%**
S10b	3× manipulation at single refresh	**0%**

Key observations. The delayed EMA provides a *two-tick immunity window*: the first two refreshes after any price spike produce exactly 0% mid-price impact (S02, ticks 4–5), because the spike enters the `last` slot on the first refresh and only propagates into `mean` on the third ($\sim$4% absorption per tick). After 14 sustained refreshes, only 22.5% of a 2× move has been absorbed, consistent with the theoretical bound from §4.6 of [1]. Inter-refresh manipulation (S10a) has exactly zero effect, as does single-tick manipulation at refresh time (S10b). Liquidity changes at constant price ratio (S06–S09) produce negligible mid-price deviation ($<$0.06%), confirming that the log-space mid tracks price independently of pool depth.

IV. 3.5.3 Spread Auto-Widening Response

Table 30 records the spread percentage for a 1-unit query before and after liquidity changes. The spread correctly reflects pool depth without manual parameterization.

Table 30: Spread response to liquidity changes (1-unit query)

Scenario	Description	Before	After
S06	Sudden 10× liquidity	2.53%	2.37%
S07	Gradual 10× liquidity	2.53%	1.98%
S08	Sudden 90% drain	2.06%	3.06%
S09	Gradual 90% drain	2.06%	2.52%

EMA smoothing of the log-space spread produces an asymmetry between sudden and gradual changes: sudden liquidity addition yields $\approx$2.37% after 3 ticks (partial absorption), while gradual addition over 12 ticks converges to $\approx$1.98%. Sudden removal produces $\sim$50% spread widening (2.06% → 3.06%), providing immediate protection against thin-pool manipulation.

IV. 3.5.4 Decimal Invariance

All scenarios produce identical behavior (to within rounding) across the 6 token configurations. The sole exception is the (6/6, 180k/50) configuration, where integer truncation in 6-decimal arithmetic produces slightly coarser values (e.g., S04 residual of 3.99% vs. 3.74% elsewhere). This confirms that the log-space design is fundamentally decimal-agnostic.

IV. 3.5.5 Strengths and Weaknesses

Strengths.

1. *Two-tick delay buffer*: The first two refreshes after a spike show 0% mid-price impact, providing strong flash manipulation resistance.
2. *Inter-refresh immunity*: Price changes between refresh windows have exactly zero effect on the oracle.
3. *Self-calibrating spread*: The spread correctly widens and tightens with pool depth changes while keeping the mid stable, without per-pair configuration.
4. *Decimal invariance*: Behavior is consistent across all token decimal combinations (6, 18, and mixed).

Weaknesses (inherent to DECAY_12HL).

1. *Slow convergence*: After a legitimate 2× price move, only $\sim$22.5% is absorbed after 14 hours, potentially enabling arbitrage against stale prices.
2. *EMA reversal lag*: After a 2× spike and full return to baseline, $\sim$15% residual remains, as the EMA retains memory of past spikes beyond their market relevance.
3. *Slow drift undertracking*: A gradual 2× move over 12 hours shows only 9.1% oracle movement, creating a persistent gap between reported and actual price.

These weaknesses are the intentional cost of heavy smoothing ($\alpha \approx 0.944$). The design prioritizes manipulation resistance over responsiveness—appropriate for a lending protocol where the cost of oracle manipulation (bad debt) exceeds the cost of lagging genuine price moves

(delayed liquidations). The hourly refresh cadence further limits the attack surface, as an attacker cannot increase the sampling rate to accelerate EMA convergence toward a manipulated price.

IV. 4 Bad Debt Risk Quantification

This appendix was originally published as a standalone companion document and is reproduced here for completeness.

IV. 4.1 Introduction

XPower Banq's log-space TWAP oracle provides robust manipulation resistance—approximately 40 hours of sustained artificial pricing are required to achieve 90% deviation [18]. However, the same smoothing that resists manipulation delays response to genuine market crashes. After an instantaneous price crash, only $\sim$50% of the log-space deviation is absorbed after 13 hourly refreshes (the 12-hour half-life). Combined with the 1-hour rate limit between oracle updates, the protocol can be 2+ hours blind during a crash.

This appendix provides a complete risk quantification framework:

1. **Oracle Lag Model** (§IV. 4.2): Formal characterization of the EMA convergence delay and the resulting phantom-healthy window.
2. **Monte Carlo Simulation** (§IV. 4.3): Jump-diffusion price paths with ETH-calibrated parameters, swept across LTV and oracle decay configurations.
3. **Analytical Bound** (§IV. 4.4): Closed-form upper bound on bad debt as a function of crash magnitude, oracle decay, and LTV.
4. **Safe Operating Region** (§IV. 4.5): Parameter configurations satisfying the <5% TVL constraint.

All simulations use fixed random seeds for reproducibility.

IV. 4.2 Oracle Lag Model

IV. 4.2.1 EMA Update Semantics

The TWAP oracle (`TWAP.sol`) performs an EMA update on each refresh. Critically, the update blends the stored mean with the *previous* observation (`last`), not the current one (`next`):

$$\overline{m}_n = \alpha \cdot \overline{m}_{n-1} + (1-\alpha) \cdot \ell_{n-1} \quad (58)$$

$$\ell_n = q_n \quad (59)$$

where $\overline{m}_n$ is the smoothed log-price, ℓ_n is the stored last observation, and q_n is the current market price (in $\log_2$ space). This creates a *one-refresh delay*: a new price enters the mean only on the *following* refresh.

IV. 4.2.2 Step-Crash Convergence

After an instantaneous crash of fraction δ at time t_0 (price drops from p_0 to $p_0(1-\delta)$), the oracle-reported price after n refreshes follows:

$$\hat{p}(n) = p_0 \cdot 2^{\overline{m}_n} \quad (60)$$

where the log-space EMA converges geometrically toward the post-crash price. The fraction of the log-space deviation absorbed by the oracle at refresh n is:

$$A(n) = 1 - \alpha^{n-1} \quad (n \geq 2), \qquad A(0) = A(1) = 0 \quad (61)$$

Due to the one-refresh delay, the oracle is completely blind for the first refresh after a crash. This log-space absorption is independent of crash magnitude δ—all crashes converge at the same geometric rate $(1-\alpha)$ per refresh. The oracle-reported price in linear space is $\hat{p}(n) = p_0 \cdot (1-\delta)^{A(n)}$ (`mean.mid` is updated independently of `mean.rel`), so linear-space absorption varies with δ. Figure 24 shows the convergence trajectories for various crash magnitudes, and Table 31 shows log-space absorption percentages for the default decay $\alpha = 0.944$ (12-hour half-life).

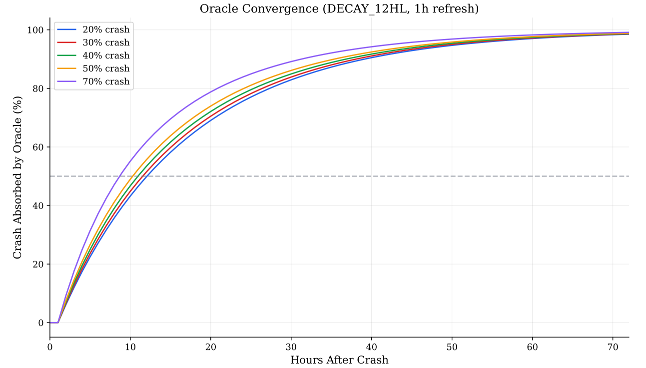


Figure 24: Oracle convergence after step crashes of various magnitudes ($\alpha = 0.944$, 1 h refresh). The dashed line marks the 12-hour half-life. All crash magnitudes converge at the same geometric rate in log-space, reaching 50% absorption near $n = 13$ refreshes (accounting for the one-refresh delay).

Table 31: Oracle absorption $A(n)$ after n hourly refreshes ($\alpha = 0.944$)

n	20%	30%	40%	50%	70%
1	0.0%	0.0%	0.0%	0.0%	0.0%
2	5.6%	5.6%	5.6%	5.6%	5.6%
4	15.9%	15.9%	15.9%	15.9%	15.9%
8	33.2%	33.2%	33.2%	33.2%	33.2%
12	46.9%	46.9%	46.9%	46.9%	46.9%
24	73.4%	73.4%	73.4%	73.4%	73.4%
48	93.3%	93.3%	93.3%	93.3%	93.3%
72	98.3%	98.3%	98.3%	98.3%	98.3%

Log-space absorption $A(n) = 1 - \alpha^{n-1}$ is independent of crash magnitude; linear-space absorption varies with δ.

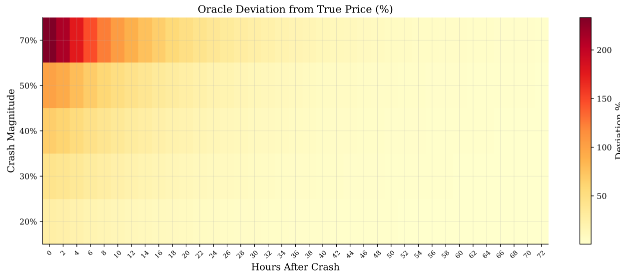


Figure 25: Oracle deviation from true price (%) as a function of crash magnitude and hours elapsed. The heatmap shows the full convergence surface; larger crashes produce higher absolute deviations but the same geometric decay rate.

IV. 4.2.3 Phantom-Healthy Windows

Definition IV. 4.1 (Phantom-Healthy Position)**.** A position is *phantom-healthy* at time t when:

$$H_{\text{oracle}}(t) = H_0 \cdot \frac{\hat{p}(t)}{p_0} \geq 1 \tag{62}$$

$$H_{\text{true}}(t) = H_0 \cdot (1 - \delta) < 1 \tag{63}$$

where H_0 is the initial health factor.

For a position to be truly underwater, the crash must satisfy $\delta > 1 - 1/H_0$. The oracle triggers liquidation when $\hat{p}(n)/p_0 < 1/H_0$, i.e., the oracle price ratio drops below the inverse of the initial health.

Definition IV. 4.2 (Phantom-Healthy Window)**.** The *phantom-healthy window* W is the number of oracle refreshes during which a position remains phantom-healthy:

$$W = \min\left\{ n \geq 0 \;\middle|\; \frac{\hat{p}(n)}{p_0} < \frac{1}{H_0} \right\} \tag{64}$$

Table 32 shows phantom-healthy windows for representative (crash, initial health) pairs with $\alpha = 0.944$.

Table 32: Phantom-healthy window W (hours) by crash magnitude and initial health, $\alpha = 0.944$

		Initial Health H_0			
Crash		1.1	1.3	1.5	2.0
20%	W	11	—	—	—
30%	W	7	25	—	—
40%	W	5	14	29	—
50%	W	4	10	17	—
70%	W	3	6	9	16

"—" indicates the position remains solvent (not underwater). Values verified against step-by-step OracleModel simulation (match within ± 1 refresh).

IV. 4.2.4 Worst-Case Blind Time

The worst case occurs when a crash happens immediately after a refresh, adding one full refresh interval of blindness. The total blind time is:

$$t_{\text{blind}} = \Delta t_{\text{refresh}} + W \cdot \Delta t_{\text{refresh}} \tag{65}$$

For the conservative-mode floor (33% LTV) configuration with positions initialized near $H_0 = 1.5$:

- **40% crash**: $W = 29$ refreshes $\Rightarrow t_{\text{blind}} = 30$ hours
- **50% crash**: $W = 17$ refreshes $\Rightarrow t_{\text{blind}} = 18$ hours
- **70% crash**: $W = 9$ refreshes $\Rightarrow t_{\text{blind}} = 10$ hours

Larger crashes produce shorter phantom windows because the true price diverges more rapidly from the oracle threshold. Positions with higher initial health (further from the liquidation boundary) require more refreshes before the oracle detects their distress.

IV. 4.3 Monte Carlo Simulation

IV. 4.3.1 Price Process

Collateral prices follow a Merton jump-diffusion model [48]:

$$\frac{dp}{p} = \mu\, dt + \sigma\, dW + J\, dN \tag{66}$$

where W is a Wiener process, N is a Poisson process with intensity λ_J, and $J \sim \mathcal{N}(\mu_J, \sigma_J^2)$ are i.i.d. jump magnitudes. Parameters are calibrated to ETH historical data (Table 33).

Table 33: Jump-diffusion parameters (ETH calibration)

Parameter	Value
Annual volatility σ	90%
Jump intensity λ_J (jumps/year)	6.0
Mean jump μ_J	-15%
Jump volatility σ_J	10%
Drift μ	compensated

IV. 4.3.2 Simulation Design

Resolution. Paths are generated at hourly resolution (8,760 steps per year), matching the oracle refresh cadence. This is a deliberate trade-off: minute-level resolution would require 100K $\times$ 525K $=$ 52.5B data points ($\sim$210 GB at float32), while hourly resolution requires only 875M points ($\sim$3.5 GB). Since the oracle can only update hourly, intra-hour price movements do not affect oracle-triggered liquidations.

Pool. Each path initializes 1,000 synthetic positions with health factors $H_0 \sim \mathcal{N}(1.5, 0.3)$ truncated at $[1.0, 3.0]$ and uniform unit borrow size.

Oracle. The log-space EMA replicates `TWAP.sol` semantics exactly, including the one-refresh delay. The oracle processes hourly price snapshots and produces oracle-price-ratio time series.

Refresh. The 1-hour cadence assumed throughout this appendix is the deployed value of the governance parameter `LIMIT_ID` (set in the deploy script and gated by the `delayed()` modifier on `Oracle.refresh()`); it is not a hard-coded constant. The conservative-mode parameter set and the phantom-window math (Eq. 65, Tab. 32) scale linearly with $\Delta t_{\text{refresh}}$ and are contingent on this cadence not being shortened materially.

Liquidation. Two models are implemented: (1) *full liquidation* via vectorized threshold crossing—for each

position, pre-compute the critical oracle price ratio $p_{\text{crit}} = 1/H_0$ and detect the first crossing with `np.argmax`; (2) *partial liquidation* that transfers fraction 2^{-e} per step (matching `Pool.sol`'s right-shift by `partial_exp`; $e = 1$ gives the default 50%-per-step), simulated step-by-step with optional dimensionless liquidation-recovery haircut κ (distinct from the cascade-simulation depth coefficient k of §IV. 2).

Parameters. Default run: 1,000 paths, LTV = 1/3 (the conservative-mode governance floor), $\alpha = 0.944$, 1-hour refresh, seed = 42. The full LTV sweep—including the shipped default of 2/3 (66.67%)—appears in Table 34; per-path detailed plots are rendered at the 1/3 floor as a worst-case visual demonstration.

IV. 4.3.3 Results

Table 34 presents Monte Carlo results across LTV configurations. The shipped default of 66.67% LTV sits at the 67% row, with $\mathbb{E}[\text{BD}] = 0.00\%$, CVaR(99%) = 0.02%, and 1.3% of paths producing any bad debt—bounded but non-zero risk under tail-event jump-diffusion paths. The conservative-mode floor of 33% LTV produces zero bad debt across all paths and metrics: the 200% over-collateralization buffer ensures the oracle triggers liquidation (at $p_{\text{crit}} = 1/H_0$) well before any shortfall accumulates. Bad debt grows non-linearly above 67%, reaching $\mathbb{E}[\text{BD}] = 0.01\%$ and CVaR(99%) = 0.69% at the 75% upper sweep point.

Table 34: Monte Carlo bad debt metrics by LTV ($\alpha = 0.944$, $N = 1{,}000$ paths)

LTV	$\mathbb{E}[\text{BD}]$	VaR(99%)	CVaR(99%)	Nonzero
33%	0.00%	0.00%	0.00%	0.0%
50%	0.00%	0.00%	0.00%	0.0%
67%	0.00%	0.00%	0.02%	1.3%
75%	0.01%	0.32%	0.69%	10.7%

All values as % of TVL per year. “Nonzero” is the fraction of paths producing any bad debt. 95% CI on $\mathbb{E}[\text{BD}]$ is $< \pm 0.10\%$ at 1,000 paths.

Table 35 shows the effect of oracle decay rate. At the 33% LTV floor, all decay rates produce zero bad debt—the over-collateralization margin dominates. Oracle decay becomes relevant only at higher LTV (including the shipped 66.67% default) where the liquidation-to-bad-debt gap is narrower.

Table 35: Monte Carlo bad debt metrics by oracle decay (LTV $= 33\%$, $N = 1{,}000$ paths)

α	HL	$\mathbb{E}[\text{BD}]$	VaR(99%)	CVaR(99%)
0.891	6 h	0.00%	0.00%	0.00%
0.944	12 h	0.00%	0.00%	0.00%
0.972	24 h	0.00%	0.00%	0.00%

HL = half-life in hours. At the 33% LTV floor, the 200% over-collateralization absorbs all oracle staleness regardless of decay rate; results at the shipped 66.67% default are sensitive to α (see Tab. 34).

IV. 4.3.4 Bad Debt Distribution

Figure 26 shows the full bad debt distribution for the conservative-mode (33% LTV) configuration. All 1,000 simulated paths produce exactly zero bad debt, confirming that the 200% over-collateralization at the 33% LTV floor fully absorbs oracle staleness. The oracle triggers liquidation at $p_{\text{crit}} = 1/H_0 \approx 0.67$ (33% price decline), well before the bad-debt threshold at $p = \text{LTV}/H_0 \approx 0.22$ (78% decline). At the shipped default of 66.67% LTV the bad-debt threshold rises to $p \approx 0.44$ (56% decline), reducing the gap and accounting for the 1.3% of nonzero-BD paths in Table 34.

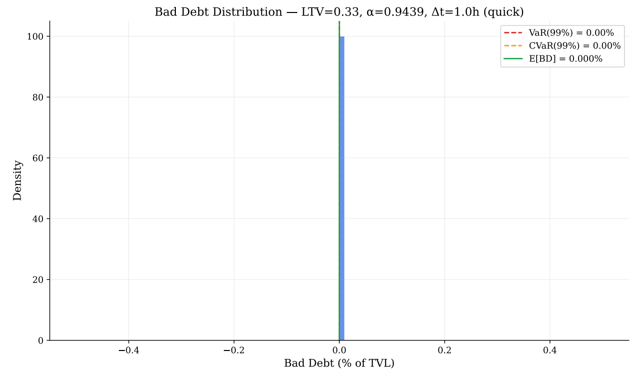


Figure 26: Bad debt distribution at the conservative-mode floor (LTV = 33%, $\alpha = 0.944$). All paths produce zero bad debt: the 200% over-collateralization ensures liquidation triggers well before any shortfall accumulates. The shipped default at 66.67% LTV produces bounded but non-zero bad debt; see Tab. 34.

IV. 4.3.5 Drawdown–Bad Debt Relationship

Figure 27 confirms that at the 33% LTV floor, even paths with extreme drawdowns (>90%) produce zero bad debt. Bad debt requires the *true* price to drop below $\text{LTV}/H_0 \approx 0.22$ *and* the oracle to fail to trigger liquidation in time—the corrected liquidation threshold ($1/H_0 \approx 0.67$) catches positions long before this point.

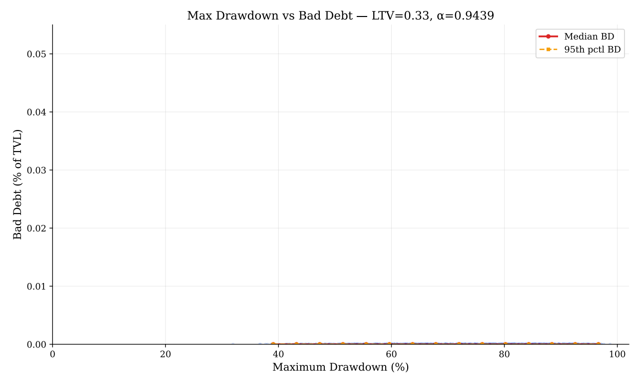


Figure 27: Maximum drawdown vs. bad debt per simulated path at the conservative-mode floor (LTV = 33%, $\alpha = 0.944$). All paths produce zero bad debt regardless of drawdown severity, confirming the analytical bound.

IV. 4.3.6 Liquidation Delay Distribution

For paths that produce liquidations, the delay between a position becoming truly underwater ($H_{\text{true}} < 1$) and the oracle triggering liquidation ($H_{\text{oracle}} < 1$) has the following characteristics (baseline configuration, 1,000 paths):

- **Median delay**: 30 hours
- **Mean delay**: >218 hours (skewed by long tails)
- **99th percentile**: ∼3,800 hours

The heavy right tail (Figure 28) reflects paths where positions hover near the liquidation boundary during gradual declines, maintaining phantom-healthy status for extended periods.

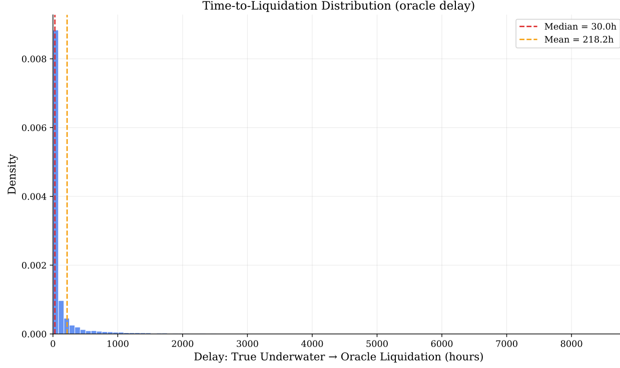


Figure 28: Distribution of oracle-induced liquidation delay (hours from true underwater to oracle-triggered liquidation). The distribution is extremely heavy-tailed: median delay is 30 h but the mean exceeds 218 h due to paths where positions hover near the liquidation boundary.

IV. 4.4 Analytical Bound

IV. 4.4.1 Instant Liquidation Baseline

As a baseline, consider bad debt under instant liquidation (no oracle delay). A position with initial health H_0 experiencing crash fraction δ has:

$$\mathrm{BD}_{\mathrm{inst}} = \mathrm{borrow} \cdot \max\left(0,\ 1 - \frac{H_0(1-\delta)}{\mathrm{LTV}}\right) \tag{67}$$

Positions are underwater when $H_0 < 1/(1-\delta)$, and produce bad debt when $H_0(1-\delta) < \mathrm{LTV}$ (shortfall exceeds collateral value at true price). Integrating over the health distribution $H_0 \sim \mathrm{TruncNorm}(1.5, 0.3, [1.0, 3.0])$ and weighting by TVL contribution:

$$\frac{\mathbb{E}[\mathrm{BD}_{\mathrm{inst}}]}{\mathrm{TVL}} = \frac{\int_1^{H_{\mathrm{crit}}} \max\left(0,\ 1 - \frac{H(1-\delta)}{\mathrm{LTV}}\right) H \cdot f(H)\, dH}{\mathbb{E}[H]} \tag{68}$$

where $H_{\mathrm{crit}} = 1/(1-\delta)$ and $f(H)$ is the truncated normal PDF.

IV. 4.4.2 Oracle Delay Penalty

The additional bad debt from oracle staleness arises when price continues declining during the phantom-healthy window. For a position with phantom window W refreshes and hourly volatility $\sigma_h = \sigma/\sqrt{8760}$, the expected additional decline is:

$$\Delta_W = W \cdot \sigma_h \approx W \cdot \frac{0.90}{\sqrt{8760}} \approx 0.0096 \cdot W \tag{69}$$

IV. 4.4.3 Conservative Upper Bound

Theorem IV. 4.1 (Bad Debt Upper Bound)**.** *For a step crash of fraction δ, oracle decay α, and effective LTV, the maximum bad debt as a fraction of TVL is bounded by:*

$$BD_{\max}(\delta, \alpha, LTV) \leq BD_{inst} \cdot \left(1 + \frac{W_{\max} \cdot \sigma_h}{\delta}\right) \tag{70}$$

where $W_{\max}$ is the phantom window for the marginal underwater position (health just below $1/(1-\delta)$) and $\sigma_h \approx 0.96\%$/hour is the ETH hourly volatility.

Proof. The bound follows from three observations: (1) all positions that are underwater under instant liquidation are also underwater under delayed liquidation, establishing $\mathrm{BD}_{\mathrm{inst}}$ as a lower bound; (2) positions with longer phantom windows experience additional price movement of order $W \cdot \sigma_h$ during the delay; (3) the marginal underwater position (highest H_0 still underwater) has the longest phantom window $W_{\max}$.

The multiplicative penalty $W_{\max} \cdot \sigma_h/\delta$ represents the worst-case fractional increase in bad debt due to continued decline during the phantom window, normalized by the initial crash magnitude. This is conservative because: (a) it assumes all positions experience $W_{\max}$ rather than their individual (shorter) windows, and (b) it assumes continued decline at full volatility rather than zero-drift random walk. □

Table 36 presents the bound for selected parameter combinations.

Table 36: Analytical bad debt bound (% of TVL)

Crash	LTV	$\mathrm{BD}_{\mathrm{inst}}$	α=0.944	α=0.891
30%	33%	0.00	0.00	0.00
30%	75%	0.06	0.20	0.13
40%	67%	0.16	0.46	0.32
40%	75%	0.97	2.79	1.90
50%	33%	0.00	0.00	0.00
50%	50%	0.00	0.00	0.00
50%	67%	1.90	5.08	3.51
50%	75%	4.98	13.32	9.20
70%	33%	0.16	0.26	0.21
70%	50%	9.67	15.52	12.73
70%	67%	29.00	46.52	38.15

$\mathrm{BD}_{\mathrm{inst}}$ is the baseline under instant liquidation. Bounds are computed with ETH hourly volatility $\sigma_h \approx 0.96\%$.

IV. 4.5 Safe Operating Region

IV. 4.5.1 Safety Criterion

We define a parameter configuration (α, LTV) as *safe* if the analytical bad debt bound satisfies $\mathrm{BD}_{\max} < 5\%$ of TVL for crash magnitudes up to 50%. This corresponds to Acceptance Criterion 4 from the specification.

IV. 4.5.2 Safe Configurations

Table 37 lists all configurations satisfying the safety criterion.

Table 37: Safe parameter configurations (BD bound $< 5\%$ for 50% crash)

α	Half-Life	LTV	BD Bound
0.891	6 h	33%	0.00%
0.891	6 h	50%	0.00%
0.891	6 h	67%	3.51%
0.944	12 h	33%	0.00%
0.944	12 h	50%	0.00%
0.972	24 h	33%	0.00%
0.972	24 h	50%	0.00%

The conservative-mode floor ($\alpha = 0.944$, LTV = 33%) produces zero bounded bad debt for any crash up to 50%. The shipped default ($\alpha = 0.944$, LTV = 66.67%) sits between the 50% and 67% rows and is governable down to 33% in one lethargic cycle (halving w_s from 170 to 85).

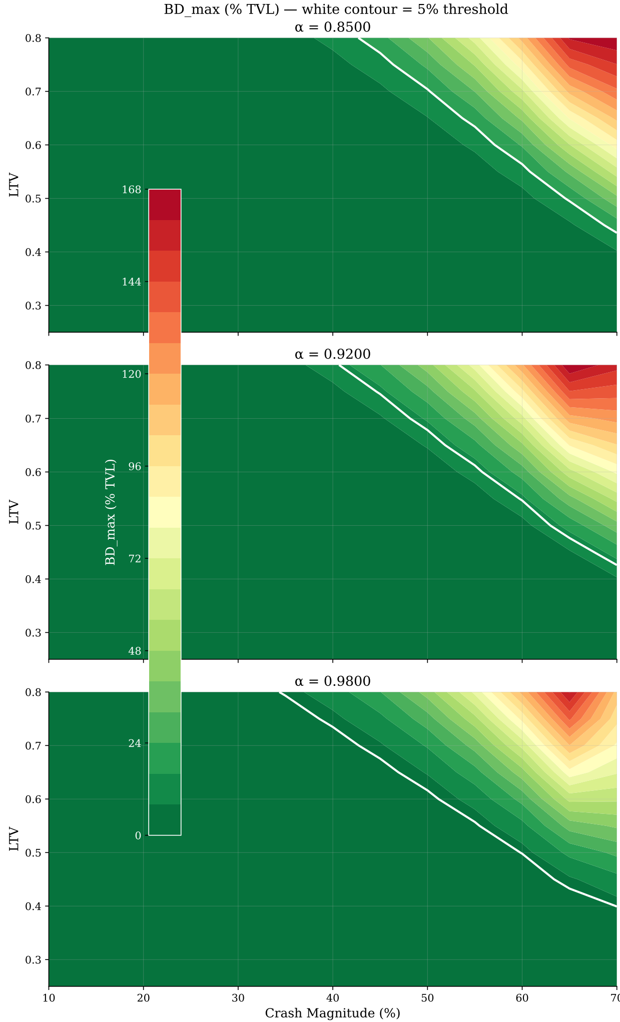


Figure 29: Safe operating region in $(\alpha, \text{LTV}, \text{crash})$ space. The surface separates configurations where $\text{BD}_{\text{max}} < 5\%$ (below) from those exceeding the threshold (above).

IV. 4.5.3 Key Observations

1. **The 33% LTV floor is zero-bad-debt for $\leq$50% crashes.** At the conservative-mode floor, positions require a >50% crash before any become underwater ($H_{\text{crit}} = 1/(1-\delta) > H_{\text{min}} = 1.0$ requires $\delta > 0$, but $H_0 \cdot (1-\delta) < \text{LTV}$ requires $\delta > 1 - \text{LTV}/H_0$). With 200% over-collateralization, even a 50% crash leaves the average position ($H_0 = 1.5$) with $H_{\text{true}} = 0.75 > \text{LTV} = 0.33$: still solvent.

2. **LTV is the dominant risk parameter.** Bad debt is zero for LTV $\leq 50\%$, bounded but non-zero at the shipped 66.67% default ($\mathbb{E}[\text{BD}] = 0.00\%$, CVaR(99%) = 0.02%), and grows non-linearly above, reaching $\mathbb{E}[\text{BD}] = 0.01\%$ and CVaR(99%) = 0.69% at the 75% upper sweep point (Table 34). The oracle decay rate has no observable effect at the 33% LTV floor (Table 35) but interacts non-linearly with LTV at the shipped default.

3. **The 50% buffer absorbs oracle staleness within tolerance at the shipped default.** The oracle triggers liquidation at $p_{\text{crit}} = 1/H_0 \approx 0.67$ (33% price decline). At the shipped 66.67% LTV, bad debt requires the true price to reach $\text{LTV}/H_0 \approx 0.44$ (56% decline)—a 23-percentage-point gap. At the conservative 33% floor, the gap widens to 45 percentage points and oracle delay is rendered harmless.

4. **Tail risk is bounded at the shipped default and negligible at the floor.** At the shipped 66.67% LTV, CVaR(99%) = 0.02%; at the conservative 33% floor, all risk metrics (E[BD], VaR, CVaR) are zero. At the 75% upper sweep point, CVaR(99%) reaches 0.69% of TVL.

IV. 4.6 Sensitivity Analysis

IV. 4.6.1 Parameter Rankings

The following parameters are ranked by their impact on expected bad debt (based on tornado diagram analysis over the MC sweep):

1. **LTV (weight ratio)** — governance-controlled, dominant effect. Moving from the 33% floor through the shipped 66.67% default to the 75% upper sweep point increases $\mathbb{E}[\text{BD}]$ from 0.00% to 0.01% and CVaR(99%) from 0.00% (at floor) through 0.02% (at default) to 0.69% (at the 75% sweep point).

2. **Jump intensity λ_J** — exogenous market condition. Higher jump frequency (12 vs. 4 per year) increases tail risk by amplifying crash-like events.

3. **Oracle decay α** — secondary effect. At the 33% LTV floor, all decay rates produce zero bad debt. At the shipped 66.67% default and above, faster decay reduces phantom-healthy windows, but the effect is dominated by the LTV choice.

4. **Refresh interval** — operational parameter. Faster refreshes (15 min vs. 2 h) reduce the initial blind period but have diminishing returns once below the half-life.

5. **Lock fraction** ϕ — emergent, not directly controlled. Locks reduce cascade-induced bad debt but have minimal effect on oracle-staleness-driven bad debt.

6. **Liquidation-recovery haircut** κ — exogenous. Increases bad debt during partial-liquidation steps in declining markets but does not affect oracle staleness. (κ here is the dimensionless MC haircut of §IV. 4.7, not the linear-depth cascade coefficient k of §IV. 2.)

IV. 4.6.2 Interaction Effects

Two notable interaction effects emerge from pairwise analysis:

$\alpha\times$ **LTV.** Higher LTV *amplifies* the staleness problem non-linearly. At the 33% LTV floor, changing α from 0.891 to 0.972 has zero observable effect; at the 75% upper sweep point, the same change measurably increases bad debt. The over-collateralization buffer acts as a threshold: below a critical LTV, oracle delay is completely absorbed; the shipped 66.67% default sits just above this threshold.

$\alpha\times$ **Refresh interval.** Diminishing returns to faster refreshes when α is already responsive. At $\alpha = 0.891$ (6 h HL), reducing refresh interval from 2 h to 15 min has minimal effect because the EMA already converges rapidly.

IV. 4.7 Partial Liquidation

The protocol uses a 2^{-e} partial liquidation model. `Pool.sol` computes `borrowed_total » partial_exp` and `supplied_total » partial_exp`, transferring fraction 2^{-e} of the position per liquidation call. Thus $e = 0$ corresponds to full liquidation (100%), $e = 1$ clears 50% per step (the default), $e = 2$ clears 25%, and increasing e liquidates a *smaller* share each step, leaving more residual debt to be wound down across subsequent calls.

During continued price declines, partial liquidation can produce *more* bad debt than instant full liquidation, because remaining position fractions face increasingly adverse prices. A dimensionless liquidation-recovery haircut κ further increases bad debt by modeling the proceeds shortfall of forced liquidation sales:

$$p_{\text{effective}} = p_{\text{true}} \cdot (1 - \kappa) \tag{71}$$

This κ is distinct from the cascade-simulation depth coefficient k of §IV. 2: κ is a dimensionless fractional haircut applied to per-step recovery, whereas k has units of 1/supply (linear-depth slippage of the cascade model). The two are not numerically comparable.

Monte Carlo tests confirm that a haircut $\kappa = 0.10$ (10%) measurably increases bad debt relative to $\kappa = 0$ in declining markets. However, the effect is secondary to the LTV choice and primarily affects already-distressed positions.

IV. 4.8 Cross-Validation

IV. 4.8.1 Oracle Model vs. Solidity

The Python oracle model is cross-validated against three Foundry scenario tests from `test/Oracle/OracleScenario.t.sol`:

- **S01 (Benign Motion)**: 24 refreshes with $\pm 5\%$ oscillations. Mid drift $< 10\%$. ✓
- **S02 (Sudden Drift)**: $2\times$ price jump. After 1st refresh: $\delta < 3\%$; after 2nd: $\delta < 15\%$; after 14th: $15\% < \delta < 70\%$. ✓
- **S04 (Sudden Reversal)**: $2\times$ spike then revert. Residual $< 5\%$ (Solidity cadence) or $< 7\%$ (Python all-tick cadence). ✓

The discrepancy in S04 arises from Solidity's `delayed()` modifier using strict $>$ comparison, causing only every other hourly tick to produce a Refresh event (alternating Pending/Refresh). Both models produce correct results for their respective refresh cadences.

IV. 4.8.2 Phantom Window Verification

Analytical phantom window computations (Section IV. 4.2.3) are verified against step-by-step OracleModel simulations for 4 representative (crash, health) pairs. All match within ± 1 refresh.

IV. 4.8.3 Bound Conservatism

The analytical bound (Theorem IV. 4.1) is verified to exceed all MC simulation results across all tested parameter combinations. The `check-bound` CLI command automates this check for each (LTV, α, crash) triple in the MC results.

IV. 4.9 Deployment Recommendations

Based on the analysis, we recommend the following parameter configurations for deployment:

1. **Shipped default** ($\alpha = 0.944$, LTV = 66.67%): Bounded bad debt of $\mathbb{E}[\text{BD}] = 0.00\%$ and CVaR(99%) = 0.02% under empirical jump-diffusion paths; analytical bound 1.90% for 50% crashes. This is the production setting, balancing capital efficiency against bad-debt containment.

2. **Conservative mode** ($\alpha = 0.944$, LTV = 33%): Zero bounded bad debt for crashes up to 50%—the governance-reachable floor for protocol-level conservatism, two lethargic cycles below the shipped default.

3. **Higher capital efficiency** ($\alpha = 0.891$, LTV = 50%): Zero bounded bad debt for 50% crashes. Requires reducing half-life to 6 hours, which weakens manipulation resistance (20 hours sustained manipulation for 90% deviation).

4. **Maximum responsiveness** ($\alpha = 0.891$, LTV = 67%): Bounded bad debt of 3.51% for 50% crash. Suitable only for highly liquid pairs where manipulation risk is low and faster oracle response is prioritized.

Configurations to avoid:

- LTV $\geq 75\%$ at any α: Bounded bad debt exceeds 9% for 50% crashes.
- $\alpha = 0.972$ (24 h HL) with LTV $> 50\%$: Excessive staleness compounds with thin over-collateralization margin.

IV. 4.10 Conclusion

XPower Banq's log-space TWAP oracle creates a quantifiable trade-off between manipulation resistance and bad-debt risk. At the shipped default of 66.67% LTV, the 50% over-collateralization buffer keeps the analytical and Monte Carlo bad-debt bounds tight: $\mathbb{E}[\text{BD}] = 0.00\%$, CVaR(99%) = 0.02%, and 1.3% of jump-diffusion paths producing any bad debt. At the conservative-mode floor of 33% LTV, the 200% over-collateralization fully absorbs oracle staleness: Monte Carlo simulation produces zero bad debt across all 1,000 simulated paths, and the analytical bound confirms zero bad debt for crashes up to 50%. Bad debt grows non-linearly above 67%, reaching $\mathbb{E}[\text{BD}] = 0.01\%$ at the 75% upper sweep point.

The mechanism underlying this robustness is the gap between the oracle's liquidation trigger ($p_{\text{crit}} = 1/H_0 \approx 0.67$, a 33% decline) and the bad-debt threshold ($p = \text{LTV}/H_0$). At the conservative 33% LTV floor, the threshold sits at ≈ 0.22 (78% decline)—a 45-percentage-point gap. At the shipped 66.67% default, the threshold rises to ≈ 0.44 (56% decline), narrowing the gap to 23 percentage points but still preserving multi-hour blind-time tolerance. Even with phantom-healthy windows of 17–30 hours for 40–50% crashes (Table 32), positions are liquidated long before the true price can traverse this gap. Only when the LTV approaches $1/H_0$—collapsing the gap—does oracle staleness translate into realised bad debt.

The primary risk lever is the LTV parameter, not the oracle decay rate. Sensitivity analysis (§IV. 4.6) ranks the six model parameters by impact: LTV dominates, followed by exogenous jump intensity λ_J, with oracle decay α a distant third. At the 33% LTV floor, all three tested decay rates ($\alpha \in \{0.891, 0.944, 0.972\}$) produce identical zero bad debt (Table 35), confirming that the over-collateralization buffer acts as a binary absorber below a critical LTV. Above that threshold (which the shipped 66.67% default crosses by a small margin), the interaction between α and LTV becomes non-linear: increasing LTV toward 75% amplifies the staleness penalty from $0\times$ to $1.67\times$ the instant-liquidation baseline.

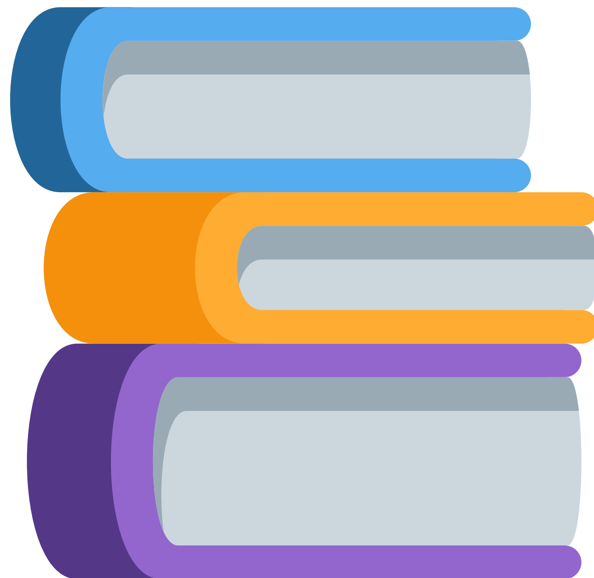
Governance should therefore prioritize LTV conservatism over oracle responsiveness. The analytical bound $\text{BD}_{\max}(\delta, \alpha, \text{LTV}) \leq \text{BD}_{\text{inst}} \cdot (1 + W_{\max} \cdot \sigma_h/\delta)$ (Theorem IV. 4.1) provides a closed-form, governance-usable formula for evaluating parameter changes against bad-debt tolerance. Combined with the safe-operating-region criterion ($\text{BD}_{\max} < 5\%$ of TVL for $\delta \leq 50\%$), this bound delineates five safe (α, LTV) configurations (Table 37), all with $\text{LTV} \leq 67\%$.

Limitations. The analysis makes several simplifying assumptions. First, positions are modelled with a static health distribution $H_0 \sim \mathcal{N}(1.5, 0.3)$ rather than endogenous dynamics from supply/borrow inflows. Second, oracle refreshes are assumed to occur at exact hourly intervals, whereas the `delayed()` modifier's strict-$>$ comparison can cause alternating Pending/Refresh patterns that effectively double the refresh interval (§IV. 4.8.1). Third, liquidation cascades and their feedback effects on market prices are modelled only via the liquidation-recovery haircut κ (§IV. 4.7), not through an order-book simulation. Fourth, correlated multi-asset crashes are not considered.

XPower Banq: Part V

References & Glossary

Consolidated Bibliography and Comprehensive Terminology for the XPower Banq Lending Protocol Suite



References

[1] Karun The Ritch. XPower Banq: A Permissionless DeFi Lending Protocol with Lethargic Governance and Beta-Distributed Position Caps. *Preprint*, 2026. https://www.xpowerbanq.com

[2] Karun The Ritch. XPower Banq: Technical Appendices. *Preprint*, 2026. https://www.xpowerbanq.com

[3] Karun The Ritch. XPower Banq: Ring-Buffer Time Locks. *Preprint*, 2026. https://www.xpowerbanq.com

[4] Karun The Ritch. XPower Banq: Log-Space Compounding Index. *Preprint*, 2026. https://www.xpowerbanq.com

[5] Karun The Ritch. XPower Banq: Mathematical Theory & Proofs. *Preprint*, 2026. https://www.xpowerbanq.com

[6] Karun The Ritch. XPower Banq: Simulations & Risk Analysis. *Preprint*, 2026. https://www.xpowerbanq.com

[7] Karun The Ritch. XPower Banq: References & Glossary. *Preprint*, 2026. https://www.xpowerbanq.com

[8] Leshner, R. and Hayes, G. Compound: The Money Market Protocol. *Compound Whitepaper*, 2019. https://compound.finance/documents/Compound.Whitepaper.pdf

[9] Aave Protocol. Aave Protocol Whitepaper V2.0. *Aave Documentation*, 2020. https://docs.aave.com/

[10] MakerDAO Team. The Maker Protocol: MakerDAO's Multi-Collateral Dai (MCD) System. *MakerDAO Whitepaper*, 2017. https://makerdao.com/whitepaper/

[11] Euler Labs. Euler Finance: Permissionless Lending Protocol. *Euler Whitepaper*, 2022. https://docs.euler.finance/

[12] Lauko, R. and Pardoe, R. Liquity: Decentralized Borrowing Protocol. *Liquity Whitepaper*, 2021. https://docs.liquity.org/

[13] Frambot, P. and Gontier Delaunay, M. Morpho Optimizer: Optimizing Decentralized Liquidity Protocols. *Morpho Whitepaper*, 2022. https://whitepaper.morpho.org/

[14] Bartoletti, M., Chiang, J.H., and Lafuente, A.L. SoK: Lending Pools in Decentralized Finance. *Financial Cryptography and Data Security (FC)*, LNCS vol. 12676, pp. 553–578, 2021.

[15] Werner, S.M., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., and Knottenbelt, W.J. SoK: Decentralized Finance (DeFi). *ACM Computing Surveys*, 2022.

[16] Gudgeon, L., Perez, D., Harz, D., Livshits, B., and Gervais, A. The Decentralized Financial Crisis. *Crypto Valley Conference*, 2020.

[17] Perez, D., Werner, S.M., Xu, J., and Livshits, B. Liquidations: DeFi on a Knife-edge. *Financial Cryptography and Data Security*, 2021.

[18] Mackinga, T., Nadahalli, T., and Wattenhofer, R. TWAP Oracle Attacks: Easier Done than Said? *IEEE International Conference on Blockchain and Cryptocurrency*, 2022.

[19] Ellis, S., Juels, A., and Nazarov, S. ChainLink: A Decentralized Oracle Network. *Chainlink Whitepaper*, 2017. https://chain.link/whitepaper

[20] Angeris, G. and Chitra, T. Improved Price Oracles: Constant Function Market Makers. *ACM Conference on Advances in Financial Technologies (AFT)*, 2020.

[21] Adams, H., Zinsmeister, N., and Robinson, D. Uniswap v2 Core. *Uniswap Documentation*, 2020. https://docs.uniswap.org/

[22] Adams, H., Zinsmeister, N., Salem, M., Keefer, R., and Robinson, D. Uniswap v3 Core. *Uniswap Documentation*, 2021. https://docs.uniswap.org/

[23] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., and Juels, A. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. *IEEE Symposium on Security and Privacy (S&P)*, pp. 910–927, 2020.

[24] Qin, K., Zhou, L., and Gervais, A. Quantifying Blockchain Extractable Value: How dark is the forest? *IEEE Symposium on Security and Privacy (S&P)*, pp. 198–214, 2022.

[25] Qin, K., Zhou, L., Livshits, B., and Gervais, A. Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit. *Financial Cryptography and Data Security (FC)*, LNCS vol. 12674, pp. 3–32, 2021.

[26] Immunefi. Hack Analysis: Beanstalk Governance Attack, April 2022. *Security Report*, 2022. https://medium.com/immunefi/hack-analysis-beanstalk-governance-attack-april-2022-f42788fc821e

[27] Glassnode Insights. What Really Happened to MakerDAO on Black Thursday. *Glassnode Analysis*, March 2020. https://insights.glassnode.com/what-really-happened-to-makerdao/

[28] Liu, J., Makarov, I., and Schoar, A. Anatomy of a Run: The Terra Luna Crash. *Harvard Law School Forum on Corporate Governance*, May 2023. https://corpgov.law.harvard.edu/2023/05/22/anatomy-of-a-run-the-terra-luna-crash/

[29] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. https://bitcoin.org/bitcoin.pdf

[30] Douceur, J.R. The Sybil Attack. *International Workshop on Peer-to-Peer Systems (IPTPS)*, LNCS vol. 2429, pp. 251–260, 2002.

[31] Dwork, C. and Naor, M. Pricing via Processing or Combatting Junk Mail. *Advances in Cryptology – CRYPTO '92*, LNCS vol. 740, pp. 139–147, 1992.

[32] Roughgarden, T. Transaction Fee Mechanism Design. *ACM Conference on Economics and Computation (EC)*, 2021.

[33] Feist, J., Grieco, G., and Groce, A. Slither: A Static Analysis Framework for Smart Contracts. *IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, 2019.

[34] Curve Finance. Vote-Escrowed CRV (veCRV). *Curve Documentation*, 2020. https://curve.readthedocs.io/dao-vecrv.html

[35] Convex Finance. Vote-Locked CVX (vlCVX). *Convex Documentation*, 2021. https://docs.convexfinance.com/convexfinance/general-information/understanding-cvx/vote-locking

[36] OpenZeppelin. TimelockController. *OpenZeppelin Contracts*, 2020. https://docs.openzeppelin.com/contracts/5.x/api/governance#TimelockController

[37] OpenZeppelin. Math Library (mulDiv). *OpenZeppelin Contracts*, 2023. https://docs.openzeppelin.com/contracts/5.x/api/utils#Math

[38] OpenZeppelin Team. OpenZeppelin Contracts. *OpenZeppelin Documentation*, 2020. https://docs.openzeppelin.com/contracts/

[39] de Bruijn, N.G. A combinatorial problem. *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen*, 49(7):758–764, 1946.

[40] Lee, P.P.C., Bu, T., and Chandranmenon, G.P. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proc. 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 78–79, 2009. https://doi.org/10.1145/1882486.1882508

[41] Razvan, P. PRBMath: Solidity Library for Fixed-Point Arithmetic. *GitHub Repository*, 2023. https://github.com/PaulRBerg/prb-math

[42] Higham, N.J. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

[43] Blanchard, P., Higham, D.J., and Higham, N.J. Accurately Computing the Log-Sum-Exp and Softmax Functions. *IMA Journal of Numerical Analysis*, 41(4):2311–2330, 2021. https://doi.org/10.1093/imanum/draa038

[44] Ethereum Foundation. Solidity Documentation: Checked or Unchecked Arithmetic. *Solidity Docs*, v0.8.x, 2024. https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic

[45] Vogelsteller, F. and Buterin, V. EIP-20: Token Standard. *Ethereum Improvement Proposals*, 2015. https://eips.ethereum.org/EIPS/eip-20

[46] Santoro, J., t11s, Jadeja, J., and Cuesta Cañada, A. EIP-4626: Tokenized Vaults. *Ethereum Improvement Proposals*, 2022. https://eips.ethereum.org/EIPS/eip-4626

[47] Kyle, A.S. Continuous Auctions and Insider Trading. *Econometrica*, 53(6):1315–1335, 1985.

[48] Merton, R.C. Option Pricing When Underlying Stock Returns Are Discontinuous. *Journal of Financial Economics*, 3(1–2):125–144, 1976.

Glossary

This glossary consolidates all terminology from the protocol whitepaper [1], the engineering primitives [3, 4], the mathematical theory [5], and the simulations [6]. Entries marked with a dagger (†) are specific to the bad-debt risk analysis ([6]).

Accrual
: The periodic update of the global index to reflect newly earned interest. In the multiplicative form, accrual multiplies the index by $\exp(r)$; in the log-space form, it adds r to the index.

Accrual Path
: The code path executed during a reindex event. Also called the *write path*, since it mutates the stored index.

Acma
: Access manager—wrapper around OpenZeppelin's `AccessManager`; manages three-tier roles per capability—the bare function-caller role (e.g. `SET_TARGET`), its `_ADMIN` (grants/revokes the caller role), and its `_GUARD` (emergency revoke). Role IDs are derived via `keccak256` of the role name and a domain string in `Roles.sol`.

$A(n)$ †
: Absorption fraction—the share of the log-space deviation absorbed after n oracle refreshes: $A(n) = 1 - \alpha^{n-1}$ for $n \geq 2$, with $A(0) = A(1) = 0$ due to the one-refresh delay.

Annualised Rate (r_{annual})
: The interest rate expressed per year in WAD. Converted to per-period yield via $r_{\text{annual}} \times \Delta t/\text{YEAR}$.

α †
: EMA decay parameter of the log-space TWAP oracle. Governs tracking speed: $\alpha = 0.5^{1/\text{HL}}$ where HL is the half-life in refreshes. Default $\alpha \approx 0.944$ (12 refreshes; with the default 1-hour refresh interval enforced by `PowLimited(1 hours)` in `Oracle.sol`, this is a 12-hour half-life).

AMM
: Automated Market Maker—decentralized exchange architecture using liquidity pools and mathematical formulas for price discovery.

APY Differential
: Difference in annual percentage yield between locked and unlocked positions; with spread s, supplier differential is $\Delta r \approx 0.09 \times r_{\text{base}}$.

Asymptotic Transition
: Parameter change mechanism where values approach targets gradually via time-weighted mean rather than discrete jumps. See Lethargic Governance.

Bad Debt
: Uncollectable debt arising when a position's collateral value falls below its debt value; protocol absorbs the loss. Quantified in [6].

Bitmap
: 16-bit mask tracking which slots contain active locks. Stored in the lower 16 bits of `cache`.

BD_{inst} †
: Bad debt under instant (zero-delay) liquidation. The baseline shortfall when a position's true collateral value $H_0(1-\delta)$ falls below LTV.

$\text{BD}_{\max}$ †
: Conservative upper bound on bad debt (Theorem IV. 4.1), incorporating the oracle delay penalty: $\text{BD}_{\max} \leq \text{BD}_{\text{inst}} \cdot (1 + W_{\max}\,\sigma_h/\delta)$.

Beta Cap
: Position limit following $12\lambda(1-\lambda)^2$ distribution where λ is the user's fraction of total supply; peaks at $\lambda = 1/3$ and vanishes at boundaries.

Bid/Ask
: Price quotes for selling (bid) vs. buying (ask) an asset; spread provides manipulation resistance.

Block Stuffing
: Attack filling blocks with transactions to delay or censor others; prevented by PoW requirements.

Breakeven Period
: Minimum holding period T^* for lock adoption to be profitable; approximately $T^* \approx D/\Delta r$ where D is secondary market discount and Δr is APY differential.

Borrow Position
: ERC20 token representing debt obligation; uses inverted transfer semantics where `transfer` pulls debt.

Borrow Rate
: The interest rate charged to borrowers, computed as $r_{\text{base}} \times (1+s)$ where s is the spread parameter. Capped at $2 \times (1+s) \times 10^{18}$.

Break-Even Ratio (R)
: The on-chain read/write ratio at which the log-space transformation is gas-neutral. Computed as $R = 1{,}200/1{,}100 \approx 1.09$.

Calculator
: Library providing overflow-safe $\log_2$ and $\exp_2$ operations for the oracle pipeline, wrapping PRB-Math's UD60x18 functions. Defines the bias constant `LOG2_ONE` $= \log_2(10^{18}) \approx 59.79$ (internal `private constant`, scaled $\times 10^{18}$ as a UD60x18 value), used to convert between raw `uint256` values and UD60x18 fixed-point representation in the paired functions `Log2()`/`Exp2()`.

Cap Floor
: Minimum position cap ensuring users can enter even when $\lambda \to 0$; prevents cold start problem.

Capital Efficiency
: Borrowing power per unit of collateral; higher LTV ratios provide greater capital efficiency.

Cascade Amplification
Ratio of actual liquidations to initial shock-induced liquidations; measures feedback loop severity. Simulated in [6].

Cascade Attenuation
Reduction of liquidation-cascade depth by factor $(1-\phi)$, where ϕ is the locked fraction of the position; the central result of Theorem I. 4.1.

Circuit Breaker
Mechanism halting cascading failures; locked positions act as circuit breakers during market stress.

Cold Start
Problem where first depositor faces zero or minimal cap due to $\lambda \to 0$; solved by cap floor mechanism.

Collateral
Assets deposited to back borrowed positions; must exceed borrow value by the over-collateralization ratio.

Compounding Index
A global accumulator tracking cumulative interest. Multiplicative form: $I = I_0 \prod \exp(r_i)$. Log-space form: $L = \sum r_i$.

Conservation
The invariant that $\texttt{totalOf}(u) = p_u \cdot \exp(L - L_u)$ holds for all users at all times, exact up to WAD rounding.

Constant Product
AMM pricing formula ($x \cdot y = k$) used to derive bid/ask quotes from reserve balances.

Coordination Game
Strategic interaction where player payoffs depend on aggregate choices; lock adoption exhibits coordination dynamics where seniority value decreases as adoption increases.

CVaR(99%) †
Conditional Value-at-Risk at the 99th percentile—the expected bad debt in the worst 1% of simulated paths.

Debt Assumption
Liquidation model where the liquidator assumes the victim's debt rather than repaying it; enables capital-efficient liquidation without requiring liquid capital.

Decay Factor
EMA smoothing parameter α; controls how quickly older price observations lose influence. See also Half-Life.

δ †
Crash fraction. An instantaneous price drop from p_0 to $p_0(1-\delta)$; e.g. $\delta = 0.50$ is a 50% crash.

Depth (Σ)
Cached epoch-weighted sum $\sum v_i(e_i+1)$ enabling $O(1)$ token-second reconstruction.

Depth Identity
The algebraic identity $D = \Sigma Q - Tt + pL$ (Theorem IIa. 5.1) that converts the $O(k)$ token-seconds sum into an $O(1)$ cached computation.

Difficulty
PoW puzzle hardness parameter; governance-adjustable per operation type to tune spam resistance.

Dual Approval
Transfer model requiring approval from both sender and receiver; used for borrow position transfers.

Dust Extraction
A theoretical attack exploiting rounding errors to extract small token amounts. Bounded at ≤ 2 wei per `totalOf` query in the log-space form.

E-fold
One unit of natural-logarithmic growth; a factor of $e \approx 2.718$. The RAY index has $\sim$115 e-folds before overflow.

Enlisting
Governance-approved process to add new tokens to a pool; subject to time delays for existing feed modifications.

EMA
Exponential Moving Average—smoothing technique giving exponentially decreasing weight to older observations. Applied in log-space for TWAP oracle.

Entry Fee
Deposit fee charged when supplying assets to a vault; accrues to existing depositors.

Epoch
Absolute quarter index $e = \lfloor t/Q \rfloor$. Each epoch spans exactly Q seconds.

ERC20
Ethereum token standard defining transfer, approve, and balance interfaces; basis for position tokens.

ERC4626
Tokenized vault standard extending ERC20 with deposit/withdraw mechanics; basis for XPower Banq vaults.

Exit Fee
Withdrawal fee charged when redeeming assets from a vault; discourages short-term liquidity cycling.

Fixed Token List
Architecture requiring predetermined token sets per pool; ensures predictable collateral requirements.

Flash Loan
Uncollateralized loan that must be repaid within the same transaction.

Formal Verification
Mathematical proof of smart contract correctness; provides stronger guarantees than testing alone.

Front-running
Inserting a transaction before a known pending transaction to profit from its price impact.

Gas Ethereum transaction execution cost; measured in gas units multiplied by gas price.

Geomean Spread
Bidirectional geometric mean of relative spreads from forward and reverse AMM queries; stored in log-space as $\log_2(1 + s_{\text{geo}})$. Provides symmetric, manipulation-resistant spread estimation.

Governance Cycle
Single parameter change period with minimum duration (e.g., monthly); bounds rate of protocol changes.

Growth Factor
The ratio $G = \exp(L - L_u)$ by which a user's principal has grown since their last snapshot.

H_0 † Initial health factor of a position: $H_0 = (\text{supply} \times w_s)/(\text{borrow} \times w_b)$. Liquidation triggers when the oracle-observed health $H_{\text{oracle}} < 1$.

Half-Life
Time for EMA weight to decay to 50%; determines TWAP responsiveness to new price observations. [1, §C] tabulates decay factors.

Hash Rate
Computational power for PoW puzzle solving, measured in hashes per second; determines solve time.

Health Check
Validation performed after operations ensuring health factor $H \geq 1$; reverts if violated.

Health Factor
Ratio of weighted supply value to weighted borrow value: $H = \sum w_s V_s / \sum w_b V_b$. Liquidation occurs when $H < 1$.

Holder-Count Scaling
Sybil resistance mechanism using the $\sqrt{n+2}$ divisor; creating accounts increases n, reducing per-account cap gains.

Holder Floor
Governable lower bound on the effective holder count used in the cap divisor; `largeHolders()` $= \max(\text{real_holders}, n_{\min})$, with $n_{\min}$ a lethargic governance parameter capped at `Constant.MIN_HOLDERS` $= 10^{18}$ (`Constant.sol:75`).

Index
Logarithmic accumulated-rate index stored in WAD (18 decimals) inside the packed state word; per-user balance growth is $\exp(I_{\text{global}} - I_{\text{user}})$, the exponential of the log-space delta. See companion paper [4] for the construction.

Initial Lock Period
Mandatory delay before first parameter change after deployment; prevents immediate manipulation.

Integrator
Library computing Δ-stamp weighted arithmetic means over (timestamp, value) tuples; accumulates area $\sum v_i \cdot \Delta t_i$ and divides by elapsed time. Used by the `Parameterized` base contract to implement asymptotic parameter transitions in lethargic governance (§6 of [1]).

Interest Rate Model
Utilization-based formula determining borrow/supply APY; slope increases sharply above kink. Formalized in [5].

Inverted Transfer
Borrow position transfer semantics where `transfer(from, amount)` pulls debt FROM the first parameter rather than pushing to it. See §4.2 of [1].

Iteration Cap
Maximum capacity gain per governance-defined period (e.g., per week); bounds accumulation rate. Simulated in [6].

Kink
Utilization threshold (e.g., 90%) where interest rate slope increases; incentivizes liquidity retention.

Lambda (λ)
Balance fraction B/S; user's holdings divided by total supply.

Large Holder
Account holding ≥ 1 full token unit; tracked for cap calculations.

Leading Zeros
PoW validation metric counting zero nibbles at start of hash; determines if difficulty threshold is met.

Lethargic Governance
Governance model with time-delayed, bounded parameter transitions. Values approach targets asymptotically, bounded to $0.5\times$–$2\times$ per governance cycle. See [1, §A].

Liquidation
Forced closure of an unhealthy position ($H < 1$); XPower Banq uses debt assumption model.

Liquidation Cascade
Destructive feedback loop where forced sales depress prices, triggering further liquidations. Modeled in [6].

Liquidation Seniority
Priority in liquidation order; locked positions gain *de facto* seniority because liquidators prefer unlocked positions for immediate liquidity.

Liquidation-Recovery Haircut (κ) †
Fraction of collateral value lost to slippage and gas during liquidation; applied to the partial-liquidation recovery model.

Liquidity Buffer
Reserve of unutilized assets available for withdrawals; maintained via optimal utilization targeting.

Lock Adoption
Aggregate fraction $\bar{\rho}$ of positions that are locked across the protocol; equilibrium adoption varies with utilization regime. Analyzed in [5].

Lock Bonus
Additional interest earned by locked suppliers; percentage of interest accrued, bounded by spread.

Lock Depth
The weighted sum $\sum v_i \times (e_i + 1)$ across a user's time-locked positions, where v_i is the locked value and e_i is the epoch. Used in lock bonus/malus computation.

Lock Malus
Interest *reduction* for locked borrowers; percentage of interest owed, bounded by spread.

Lock Ratio (ρ)
Fraction of position that is locked; $\rho = \text{lock}/\text{balance} \in [0, 1]$.

Lock Yield
The `rate` $\times$ `depth` $\times$ $(\exp(\Delta L) - 1)/(10^{18} \times$ `LOCK_TIME`$)$ bonus or malus applied to locked positions, where `LOCK_TIME` is the maximum lock horizon in seconds (16 quarterly slots $=$ 48 months $\approx 1.26 \times 10^8$ s; `Lock.sol:62`).

LOCK_TERM (Q)
One quarter $\approx$ 91.3 days—the epoch duration and ring-buffer granularity.

LOCK_TIME (L)
$16 \times Q \approx$ 48 months—maximum lock duration and permanent lock depth cap.

Log-Normal Distribution
Statistical distribution where logarithm is normally distributed; used to model position sizes in simulations ([6]).

Log-Space Index (L)
The cumulative sum of per-period WAD yields, stored in `uint256`. Initialised to 0. Grows linearly (additively).

Log-Space Oracle
Oracle architecture storing prices as $\log_2(\text{price})$ and spreads as $\log_2(1 + s)$; enables EMA smoothing as geometric-mean temporal averaging.

Log-Sum-Exp
The numerical identity $\sum \log x_i = \log \prod x_i$, used classically to avoid overflow in iterated products. The theoretical basis for the log-space index.

LTV Loan-to-Value ratio—maximum borrowing power as fraction of collateral value; default $w_s/w_b = 170/255 \approx 66.67\%$.

Market Depth
Total volume that can be traded before significantly moving price; inverse of market impact coefficient.

Market Impact
Price change from selling assets; coefficient k relates volume to price depression.

Market Impact Coefficient
Constant k in linear price impact model $\Delta p = k \cdot V$; relates sell volume to price depression. Used in cascade simulation ([6]).

Memory Decay
Weight λ^n retained by historical price observations in EMA after n refresh periods; after half-life h periods, weight decays to 50%.

Mempool
Transaction waiting area before block inclusion; PoW prevents flooding attacks against mempool.

Merton Jump-Diffusion
Asset-price model with continuous Brownian motion plus a Poisson-driven jump component; basis for the Monte Carlo bad-debt simulation.

MEV
Maximal Extractable Value—profit available from transaction ordering, insertion, or censorship.

Modular Arithmetic
Solidity's `unchecked` arithmetic where values wrap at 2^{256}. The additive log-index is compatible with modular subtraction for computing ΔL.

Monotonicity
The property that $L(t)$ is non-decreasing: each accrual adds a non-negative yield, so the index never decreases.

Monte Carlo Simulation
Randomized numerical method using many simulated paths to estimate statistical distributions; used for bad-debt risk quantification ([6]) and TWAP analysis ([6]).

Multiplicative Bounds
Constraint limiting parameter changes to $0.5\times$–$2\times$ per governance cycle; prevents rapid manipulation.

Multiplicative Index (I)
The running product of exponential growth factors, stored at RAY (10^{27}) precision. Grows exponentially.

Nash Equilibrium
Stable strategic state where no player can improve their payoff by unilaterally changing strategy; lock adoption exhibits utilization-dependent equilibria. Analyzed in [5].

Nonce
: Random value in PoW puzzle; combined with transaction data must hash below difficulty target.

Observation Window
: Time available for detecting suspicious activity during gradual capacity accumulation; enabled by iteration caps.

Optimal Utilization
: Target utilization U^* (e.g., 90%) where interest rate curve has its kink; balances efficiency and liquidity.

Oracle Aggregation
: Combining prices from multiple feeds (TraderJoe, Chainlink) using log-space EMA smoothing with bidirectional geomean spread computation.

Overflow Horizon
: The time until a stored value exceeds 2^{256}. For the multiplicative RAY index: $\sim$29–1,154 years depending on rate. For the log-space WAD index: $\sim 10^{58}$ years.

Over-collateralization
: Requirement that collateral value exceed borrow value; enforced via health factor $H > 1$.

Partial Liquidation
: Liquidation of 2^{-e} fraction of positions rather than full liquidation.

p_{crit} †
: Critical oracle price ratio that triggers liquidation: $p_{\text{crit}} = 1/H_0$. The oracle fires when $\hat{p}(n)/p_0 < p_{\text{crit}}$.

Permanent Lock
: Irrevocable lock with `dt_term` $= 2^{256}-1$. Stored in the upper 120 bits of `cache` (`cache.perma`), not in a ring slot. Contributes $p \cdot L$ to token-seconds at query time.

Phantom-healthy †
: A position state where the oracle reports $H_{\text{oracle}} \geq 1$ (solvent) but the true health $H_{\text{true}} < 1$ (underwater). Arises from oracle staleness during crashes (Definition IV. 4.1).

Pool
: Main lending/borrowing contract managing supply, borrow, settle, and redeem operations with health checks.

Pool-to-Depth Ratio
: Pool size as fraction of market depth; determines cascade severity under price shocks.

Position Lock
: Fraction ϕ of a position restricted from redemption or sale; prevents liquidation cascades.

Position Transfer
: Movement of supply or borrow position tokens between accounts. Supply uses standard ERC20 push; borrow uses inverted pull semantics. Formalized in §4.2 of [1].

PoW
: Proof-of-Work—computational puzzle required for certain operations to prevent spam.

PRBMath
: A Solidity library providing fixed-point `exp()`, `ln()`, `mul()`, and related functions at WAD (10^{18}) precision. The `exp()` function reverts when its input exceeds 133.08×10^{18}.

Price Feed
: External data source providing asset prices; XPower Banq supports TraderJoe and Chainlink feeds.

Price Shock
: Sudden price change used to test TWAP responsiveness; EMA smoothing dampens shock impact based on half-life configuration.

Principal
: Base position amount before interest accrual; multiplied by index ratio to get current balance.

Protocol Margin
: Revenue retained by protocol from interest spread; $M(\bar{\rho}) = 2s(1 - \bar{\rho})$ where s is spread and $\bar{\rho}$ is lock adoption rate.

Protocol Parameters
: Governable constants (weights, decay, spread, rates, caps) that control protocol behavior; each transitions lethargically. Enumerated in [1, §A].

Quote / TWAP packed word
: Log-space price representation packed into a `uint256` word with four fields: `mid` ($\log_2(\text{price} \times 10^{18})$, biased by `LOG2_ONE`), `rel` ($\log_2(1 + s_{\text{geo}})$ where s_{geo} is the bidirectional geomean spread), `utc` (timestamp), and `dec` (decimal-pair packed as `(dec_source « 8) | dec_target` for unit normalisation). The containing struct is `TWAP { uint256 last; uint256 mean; }` with two such words per pair: `last` for the most recent observation and `mean` for the EWMA running mean. Replaces linear bid/ask pairs with a compact log-space encoding suitable for EMA smoothing.

Rate Limit
: Token-bucket mechanism bounding operation frequency; configured per pool with capacity and regeneration rate.

Ray
: Fixed-point representation with 27 decimal places (10^{27}).

Read Path
: The code path executed when querying a user's balance via `totalOf`. In the log-space form, this is where `exp()` is called.

Reentrancy Guard
: Protection preventing a contract from being called recursively during execution; uses transient storage.

Reindex
: Interest compounding mechanism that updates the global log-space index and snapshots per-user indices. The functions `_reindex` and `_reindexWith` advance the global log-space index; lock bonus/malus is applied per-user via `_spreadDiff` consumed by the IR model when computing the position rate. Per-user balance growth is $\exp(I_{\text{global}} - I_{\text{user}})$.

Reserves
: Token balances held in AMM liquidity pools; used to calculate bid/ask quotes via constant product formula.

Ring-Lock
: Base mechanism: ring buffer + bitmap + cached total. 9 words per user.

Role Guard
: Contract restricting which addresses can execute role-gated functions; part of access control system.

Sandwich Attack
: MEV attack bracketing a victim transaction with front-run and back-run to extract value.

Secondary Market Discount
: Price reduction D (typically 3–10%) for locked position tokens relative to NAV; reflects inability to redeem for underlying assets.

Self-Healing
: `more()` correcting stale slot contributions during overwrite, maintaining `total` and `depth` consistency per-slot (Theorem IIa. 6.7).

σ, σ_h †
: Annualized and hourly volatility of the collateral asset. $\sigma_h = \sigma/\sqrt{8760}$. ETH calibration: $\sigma = 90\%$, $\sigma_h \approx 0.96\%$/hour.

Slippage
: Price deviation from executing a trade against AMM reserves; increases with trade size relative to reserves.

Solvency Boundary
: Parameter constraint ensuring protocol can meet all obligations; default configuration satisfies $r_{\text{bonus}} + r_{\text{malus}} = 2s$, maintaining solvency for any lock adoption rate.

Spread
: (1) Oracle: bidirectional geomean of rel. spreads from both AMM query directions, stored as $\log_2(1 + s_{\text{geo}})$; wider spreads indicate lower liquidity or higher manipulation resistance. (2) IRM: symmetric half-spread parameter (e.g., 10%) applied to base rate; borrow rate = base $\times$ $(1{+}s)$, supply rate = base $\times$ $(1{-}s)$.

Spread Scaling
: Logarithmic widening of bid/ask spreads for large positions via $\mu = \log_2(x{+}1){+}1$ where $x =$ center$\cdot s$ is the notional spread (equivalently $\log_2(2x + 2)$); reflects market impact without governance parameters.

Square
: Restricted-access liquidation function executing debt assumption: for a given exponent e, atomically transfers $\lfloor \text{borrow}/2^e \rfloor$ and $\lfloor \text{supply}/2^e \rfloor$ from victim to liquidator via bit-shift. Reverts if the victim is healthy (`wnav_supply >= wnav_borrow`, equivalently $H \geq 1$, treating the boundary as solvent) or the liquidator's post-transfer health is insufficient. See also Debt Assumption, Partial Liquidation.

Staleness
: Age of price data from an oracle; stale prices beyond threshold are rejected to prevent exploitation.

Supply Position
: ERC20 token representing deposited collateral; uses standard transfer semantics.

Supply Rate
: The interest rate earned by suppliers, computed as $r_{\text{base}} \times (1 - s)$ where s is the spread. Always less than or equal to the base rate.

Sybil Attack
: Creating multiple accounts to gain unfair advantage; XPower Banq defends against rapid capacity monopolization.

Sybil Resistance
: Protection against Sybil attacks; in XPower Banq, bounds accumulation *rate* via holder-count scaling, not equilibrium share.

Time-lock
: Mandatory delay between proposing and executing governance parameter changes; prevents sudden malicious updates.

Time-Lock (Lock Extension)
: Extension of Ring-Lock adding the `depth` mapping for token-seconds. 10 words per user.

Time-Weighted Mean
: Integration technique averaging parameter values over time; enables smooth transitions without discrete jumps.

Token Bucket
: Rate limiting mechanism with capacity C, regeneration rate, and per-operation cost; operation allowed iff $C \geq 0$.

Token-Seconds
: $\sum v_i \times$ remaining time—integral of locked amount over remaining time. The depth metric that drives graduated commitment.

Truncation Bias
: The systematic negative error introduced by fixed-point truncation (rounding toward zero). In the multiplicative form, this bias compounds over N accrual

steps ($\leq 2N$ ULP). The log-space form has zero truncation during accrual.

TVL †
Total Value Locked—the aggregate collateral value in the lending pool. All bad-debt metrics are reported as a percentage of TVL.

tx.origin
Transaction originator address included in PoW hash; prevents front-runners from reusing others' solutions.

TWAP
Time-Weighted Average Price—price smoothed over time via log-space EMA (geometric mean temporal averaging) to resist manipulation. Simulated in [6].

UD60x18
PRBMath's unsigned 60.18-decimal fixed-point type. 60 integer digits and 18 fractional digits, fitting in `uint256`. Used for `exp()`, `mul()`, and `ln()` operations.

uint256
Solidity's 256-bit unsigned integer type, holding values from 0 to $2^{256} - 1 \approx 1.16 \times 10^{77}$. The storage type for both the multiplicative and log-space indices.

ULP Unit in the Last Place—the smallest representable increment at a given precision. At WAD: 1 ULP = 1 wei = 10^{-18}.

User Snapshot (L_u)
The value of L at the user's last state transition, stored in `_userIndex[user]`.

Utilization
Ratio of borrowed assets to supplied assets in a vault; drives interest rates via a piecewise-linear model with a kink at optimal utilization.

VaR(99%) †
Value-at-Risk at the 99th percentile—the bad debt level exceeded in only 1% of simulated paths.

Vault
ERC4626-compliant custody contract holding deposited assets; tracks utilization for the interest rate model.

W † Phantom-healthy window—the number of oracle refreshes during which a position remains phantom-healthy after a crash (Definition IV. 4.1).

WAD
Fixed-point representation with 18 decimal places (10^{18}).

Weight
Multiplier applied to asset values in health calculations; determines effective LTV. Default: $w_s = 170$, $w_b = 255$.

Write Path
The code path executed during index accrual (`_reindex`). In the log-space form, this is a single addition; in the multiplicative form, it calls `exp()` and `mul()`.