

WHITE PAPER · VERSION 1.0

Bunch: Covenant-Enforced Group Buying with Early Refund Bonuses on Bitcoin Cash

A dominant assurance contract for small shops: hit the group size and the shop is paid; miss it and every buyer is refunded, early joiners with a bonus. CashScript covenants hold the money and any open page settles.

Arpit Singh · GitHub LSUDOKO

November 2026 · Version 1.0

BCH BLAZE 2026 · Mainnet Track

Abstract

Group buying has a coordination problem and a custody problem. Each buyer waits for others to commit before joining, and whoever collects the money must be trusted to refund it if the group never forms. Bunch removes both problems on Bitcoin Cash (BCH). A pair of CashScript covenants holds every satoshi. If the group reaches its target size, any open copy of the web page broadcasts a keyless transaction that pays the shop and an exact platform fee. If the deadline passes first, any page broadcasts a keyless refund that returns each buyer's payment and adds a bonus for early joiners, paid from a pool the shop locked when it created the offer. The early bonus turns the offer into a *dominant assurance contract* (Tabarrok, 1998): for an early buyer, joining is no worse than staying out whatever the others do, provided the bonus exceeds the buyer's own transaction fee and the refund is broadcast before the grace period ends. Laboratory studies report that refund bonuses "can increase the rate of campaign success by 50% or more" and that bonuses limited to early contributors work as well while paying out less (Cason, Tabarrok & Zubrickas, 2021).

The system has no backend that holds keys or money. The full offer specification travels in the URL fragment; state is read from Electrum servers; wallets sign over WizardConnect and the page broadcasts. Authority without message signing comes from spending CashTokens NFTs: a buyer's receipt authorises leaving, and a shop key authorises accepting a smaller group. The contracts are covered by 131 transaction tests on the real BCH virtual machine (20 valid flows, 111 single-change attacks), mutation testing (82 of 95 rule mutants and 33 of 35 operator mutants killed, every survivor explained), metamorphic and property-based tests. Join, leave, complete, refund and merchant-accept ran on testnet3; on mainnet a real phone wallet (Selene) joined a bunch and the open page settled it keylessly, paying a platform fee of exactly 1,200 satoshis (2% of 60,000). The contracts have not had a third-party audit, and Bunch has no paying shops yet.

Contents

1. Introduction 2
2. Background 3
3. Design goals and threat model 4
4. Protocol 5
5. Economic analysis 8
6. Implementation 10
7. Security analysis 11
8. Evaluation and on-chain evidence 14
9. Limitations 15
10. Roadmap 16
11. Conclusion 16
References 17
Appendix A. Contract function reference 17
Appendix B. Glossary 18

1. Introduction

1.1 The problem

A group deal is a conditional promise: a shop offers a lower price if enough buyers commit. Three frictions keep such deals rare among small shops and informal groups.

1. **First-mover risk.** A deal needs a quorum, but each buyer would rather wait and see whether others join. Joining early carries the risk that the group never forms and money sits idle or is lost. Threshold campaigns stall for the same reason.
2. **Custody risk.** Someone has to hold the money between the first commitment and the outcome. In a chat-group order that is one organiser; on a deal platform it is the platform. Either must be trusted to pay the shop on success and to refund everyone on failure.
3. **Platform cost.** Intermediaries that solve custody charge for it. The project's research notes record a 2011 analysis of Groupon merchant agreements in which Groupon kept about 42% of revenue on average; this figure comes from a secondary summary and the primary source is not linked in our notes, so we cite it only as an indication of scale.

1.2 Contribution

Bunch combines a known economic mechanism with a programmable money system that can enforce it without an operator.

- **A dominant assurance contract as a UTXO covenant.** The shop's bonus pool is locked on chain at creation. If the group misses its deadline, the covenant pays each early buyer their payment plus a fixed bonus. The promise is enforced by consensus rules, not by the shop's goodwill (Section 4).
- **Keyless settlement by any open page.** The `complete` and `refund` paths need no signature. Every open copy of the page checks the chain and broadcasts when the rules allow. Duplicate broadcasts conflict and fail harmlessly. Exact-output rules stop the broadcaster from moving money between recipients (Finding F1, Section 7).
- **Authority by spending an NFT.** The wallet protocol used, WizardConnect, signs transactions only. Bunch proves "this is my seat" by spending the buyer's receipt NFT, and "I am the shop" by spending a shop key NFT.
- **The offer lives in the link.** Every contract parameter is encoded in the URL fragment, so any browser can rebuild both contract addresses and read state from public Electrum servers. There is no database.
- **Early rebates survive a smaller deal (Decision D1).** If the shop accepts fewer buyers than planned, early joiners still receive the bonus, as a rebate.
- **Mutation-tested covenants.** Every `require` in the contracts is removed in turn; the test suite must notice.

1.3 Scope and status

Bunch is a working application deployed on BCH mainnet at https://bunch-cash.pages.dev, built for BCH BLAZE 2026 (Mainnet Track). It started from a planning and build pack (contracts, 103 VM tests, mutation script and design documents) imported in the first commits; Finding F1, Decision D1, 28 further VM tests, the web application, the transaction builders, wallet integration and the on-chain runs were built on top of it. There are no users, paying shops, partners or revenue yet. Everything in this paper describes the implementation in the public repository.

2. Background

2.1 Assurance contracts and the dominant assurance contract

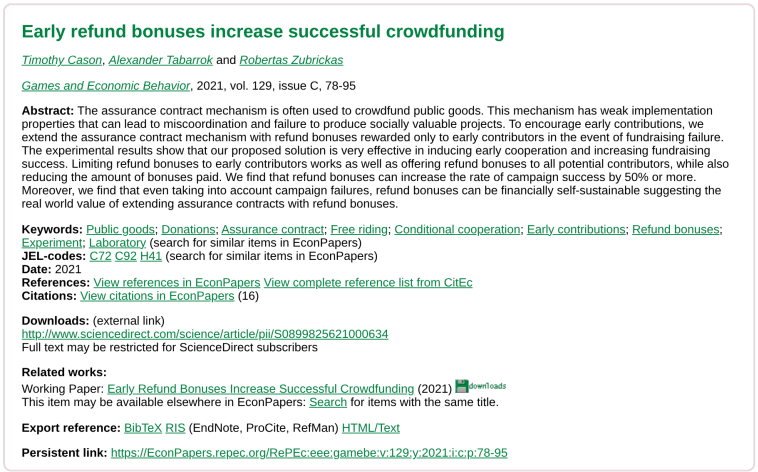
An *assurance contract* collects pledges toward a threshold: if the threshold is met, the pledges are collected and the good is provided; if not, pledges are returned. It removes the risk of paying for nothing, but it does not make contributing attractive. A rational participant who expects the campaign to fail gains nothing by pledging.

Tabarrok (1998) proposed the **dominant assurance contract**. The entrepreneur adds a payment to contributors if the campaign fails. Each contributor then either receives the good (on success) or receives their money back plus a bonus (on failure). Contributing becomes a dominant strategy: it is at least as good as abstaining whatever the others do. The entrepreneur bears the cost of the bonus only when the campaign fails, which in equilibrium it should not.

2.2 Experimental evidence on refund bonuses

A line of laboratory work by Cason, Zubrickas and Tabarrok tests the idea.

- Cason and Zubrickas (2017) find that refund bonuses improve threshold fundraising in the lab; Cason and Zubrickas (2019) extend the design to donation-based crowdfunding.
- Cason, Tabarrok and Zubrickas (2021), *Early refund bonuses increase successful crowdfunding*, report that refund bonuses "can increase the rate of campaign success by 50% or more", and that limiting bonuses to **early** contributors works as well as bonuses for everyone while paying out less (Figure 1). Bunch adopts this early-only design: only the first `earlyK` seats earn the bonus.
- Cason, Tabarrok and Zubrickas (2025), *Signaling Quality*, find that a refund bonus also acts as a credible quality signal: in their experiment bonuses were offered more often on good projects and backers chose bonus projects more often. For a shop, the bonus badge is therefore also marketing.



Early refund bonuses increase successful crowdfunding

Timothy Cason, *Alexander Tabarrok* and *Robertas Zubrickas*

Games and Economic Behavior, 2021, vol. 129, issue C, 78-95

Abstract: The assurance contract mechanism is often used to crowdfund public goods. This mechanism has weak implementation properties that can lead to miscoordination and failure to produce socially valuable projects. To encourage early contributions, we extend the assurance contract mechanism with refund bonuses rewarded only to early contributors in the event of fundraising failure. The experimental results show that our proposed solution is very effective in inducing early cooperation and increasing fundraising success. Limiting refund bonuses to early contributors works as well as offering refund bonuses to all potential contributors, while also reducing the amount of bonuses paid. We find that refund bonuses can increase the rate of campaign success by 50% or more. Moreover, we find that even taking into account campaign failures, refund bonuses can be financially self-sustainable suggesting the real world value of extending assurance contracts with refund bonuses.

Keywords: Public goods; Donations; Assurance contract; Free riding; Conditional cooperation; Early contributions; Refund bonuses; Experiment; Laboratory (search for similar items in EconPapers)
JEL-codes: C72 C92 H41 (search for similar items in EconPapers)
Date: 2021
References: View references in EconPapers View complete reference list from CitEc
Citations: View citations in EconPapers (16)

Downloads: (external link)
http://www.sciencedirect.com/science/article/pii/S0899825621000634
Full text may be restricted for ScienceDirect subscribers

Related works:
Working Paper: Early Refund Bonuses Increase Successful Crowdfunding (2021)
This item may be available elsewhere in EconPapers: Search for items with the same title.

Export reference: BibTeX RIS (EndNote, ProCite, RefMan) HTML/Text

Persistent link: https://EconPapers.repec.org/RePEc:eee:gamebe:v:129:y:2021:i:c:p:78-95

Figure 1. Abstract page of Cason, Tabarrok & Zubrickas (2021), *Games and Economic Behavior* 129:78–95, as published on EconPapers (screenshot taken Nov 22, 2026).

These are laboratory results with stylised payoffs. They motivate the design; they do not show that Bunch will raise the success rate of real group orders, which remains to be measured.

2.3 Group-buying literature

- **Anand and Aron (2003)** study optimal group-buying price schedules under demand uncertainty. Their analysis motivates tiered prices, which are on the Bunch roadmap (Section 10).
- **Jing and Xie (2011)** show that group discounts can turn informed buyers into sales agents who recruit others, and that group buying can beat referral rewards when sharing through social networks is efficient. Bunch relies on share links rather than separate referral payments.

- **Hu, Shi and Wu (2013)** compare simultaneous and sequential group-buying mechanisms and find that telling later buyers how many have already joined increases purchases. Bunch shows a live counter (one grape per seat) updated from chain subscriptions.
- **Pinduoduo's** team purchase, described in an industry interview and a Y Combinator write-up, pairs a solo price with a team price, a short window and teams as small as two. Bunch aims for small groups and short deadlines.

2.4 Bitcoin Cash covenants, introspection and CashTokens

BCH script supports **native introspection**: a script can read the values, locking scripts and token data of the transaction's inputs and outputs. A *covenant* uses these opcodes to restrict how its coins may be spent, for example "output 0 must pay this address at least this many satoshis". **CashScript** is a high-level language that compiles to BCH script; Bunch uses CashScript and its compiler `cashc` pinned at 0.14.0. Contracts are deployed as pay-to-script-hash with 32-byte hashes (P2SH32).

CashTokens add native fungible tokens and non-fungible tokens (NFTs) to BCH outputs. A token *category* is the transaction ID of the outpoint spent as input 0 of the genesis transaction, and it must be a vout-0 outpoint. An NFT carries a *commitment* (a short byte string) and a *capability*: immutable, mutable or minting. A minting NFT can create new NFTs of its category. Covenants can check token categories, commitments and capabilities through introspection, which lets a contract maintain state in a commitment and carry it from transaction to transaction (a *state thread*).

3. Design goals and threat model

3.1 Goals

Goal	How Bunch meets it
G1. No custody	Two covenants hold all funds; the site and its single server function hold no keys
G2. No operator for settlement	`complete`, `refund`, `escape` and `reclaim` need no signature; any page may broadcast
G3. The promise to early buyers is enforced	The bonus pool is locked at genesis; `refund` and `merchantAccept` must pay it
G4. Exact accounting	Fee, refund and rebate outputs are exact; only the miner fee is variable, capped by `maxFee`
G5. Liveness without the site	After `graceEnd`, `escape` returns each seat and `reclaim` returns the pool
G6. One approval for a buyer	A join is one transaction with only the buyer's inputs signed
G7. Plain UI	A judge or shopper understands the home page in seconds

3.2 Trust assumptions

- BCH consensus and the CashScript compiler (`cashc` 0.14.0) behave as documented.
- Electrum servers report UTXOs honestly. A lying server can show the wrong state or hide a bunch, but it cannot move money; the app refuses to offer Join when the chain does not match the link.
- The wallet signs only the inputs listed in the sign request and leaves contract inputs' unlocking bytecode as built.
- The shop delivers the goods. The covenant guarantees payment rules, not delivery.
- The person who shares a link chose its parameters. The app validates them, but a buyer should read the price, group size and deadline before joining.

3.3 Adversaries and attack surfaces

We consider an adversary who can build arbitrary transactions, mint tokens in categories they control, park outputs at the contract addresses, craft bunch links, choose the miner fee on keyless paths, and race other broadcasters. The attack surfaces are: the contract functions; crafted links; forged tokens from other categories; keyless broadcasts; the wallet signing channel; and chain reads (dust sent to contract addresses, indexer gaps, network mismatch). The threats and their mitigations are listed in Section 7.

4. Protocol

4.1 Notation

Symbol	Spec field	Meaning
`p`	`price`	satoshis per seat
`T`	`target`	seats needed to complete
`C`	`cap`	maximum seats (at most 50 in the app)
`K`	`earlyK`	seats with seat number `0 … K-1` earn the bonus
`b`	`bonusEach`	bonus per early seat, in satoshis
`ψ`	`feeBps`	platform fee in basis points
`D`	`deadline`	refunds open at or after this locktime
`G`	`graceEnd`	after this, seats can escape and the pool can be reclaimed
`M`	`maxFee`	maximum miner fee on keyless paths
`n`	state commitment	current number of live seats (`count`)

Seat numbers are 0-based in the contract and shown to users as 1-based ("Seat 1" is seat number 0).

4.2 The specification lives in the link

A bunch is fully described by a JSON object encoded as `v1.<base64url(JSON)>` in the URL fragment (`/b#v1.…`). It contains the network, the token category, the parameters of Section 4.1, the shop's and the fee recipient's locking bytecode, and display metadata (title, shop name, photo link, solo price, pickup note). The fragment is never sent to a server. Any browser decodes it, validates it (`validateSpec` in `src/core/spec.ts`), compiles both contract addresses and reads state from Electrum. Metadata is display-only and treated as untrusted.

Validation enforces, among other rules: `p ≥ 5,000` sats; `1 ≤ T`, `1 ≤ C ≤ 50`; `0 ≤ K ≤ C`; `b = 0` or `b ≥ 546`; `5b ≤ p` (bonus at most 20% of the price); `0 ≤ ψ ≤ 1,000` (at most 10%); `D` and `G` of the same kind with `G > D`; a grace period of at most 30 days (4,320 blocks for height deadlines); `M ≥ 2,000`; and `M ≥ maxFeeFor(C) - 1,000`.

4.3 Two covenants

BunchState (13 parameters: `bunchCategory`, `seatLB`, `merchantLB`, `feeLB`, `p`, `T`, `C`, `K`, `b`, `ψ`, `D`, `G`, `M`) is the state thread. Its UTXO holds the bunch's minting NFT and the shop's bonus pool as its BCH value. It implements `join`, `complete`, `refund`, `leave`, `merchantAccept` and `reclaim`.

Seat (`bunchCategory`, `G`, `M`) holds one buyer's payment and an immutable seat NFT. It has two functions. `withState` only requires that input 0 holds the bunch's minting NFT, delegating every other rule to BunchState. `escape` lets anyone, after `G`, send the seat alone to the refund address stored in its commitment.

Address derivation has no circular dependency: the Seat locking bytecode `seatLB` is computed first from `(bunchCategory, G, M)` and passed as a parameter to BunchState. The category is passed in **reversed** byte order relative to the hex that explorers display; the tests prove this.

4.4 Token layout

Each bunch is its own CashTokens category.

Token	Capability	Lives at	Commitment
State thread	minting	BunchState	`01` · count (u16 LE) · flags (u8) · nextSeat (u16 LE)
Seat	immutable	Seat covenant	`53` · seatNo (u16 LE) · buyer's refund locking bytecode
Receipt / voucher	immutable	buyer's token-aware address	`52` · seatNo (u16 LE)
Shop key	immutable	shop's wallet	`4d`

Genesis creates the state with commitment `010000000000` (count 0, flags 0, nextSeat 0). Each join increments both `count` and `nextSeat`; each leave decrements `count` and leaves `nextSeat` unchanged, so seat numbers are never reused and early seats cannot be re-issued. The flags byte is carried unchanged and currently unused. Receipt and shop-key outputs carry 1,000 sats; plain outputs are at least 546 sats.

4.5 Transactions

All indices are input or output positions. "Anyone" means no signature is required.

Transaction	Who	Inputs	Outputs
prep (app)	shop	shop coins	[0] shop coin at vout 0, change
genesis (app)	shop	[0] vout-0 coin, more coins	[0] state (minting NFT + pool), [1] shop key `4d`, [2] OP_RETURN (BCMR hash + URI), [3] change
`join`	buyer, one signature	[0] state, [1..] buyer's plain BCH	[0] state′, [1] seat (= `p`), [2] receipt, [3] optional change
`leave`	buyer, spends own receipt	[0] state, [1] seat, [2] receipt, [3..] fee coins	[0] state′, [1] refund to seat's stored address, [2] optional change
`complete`	anyone	[0] state, [1..n] every seat	[0] shop, [1] platform fee
`refund`	anyone, at or after `D`	[0] state, [1..n] every seat	[0] shop's leftover pool, [i] buyer of seat input i
`merchantAccept`	shop, spends key NFT	[0] state, [1] key, [2..] every seat	[0] shop, [1] fee, [2..] one rebate per early seat
`escape`	anyone, after `G`	[0] one seat	[0] that seat's buyer
`reclaim`	anyone, after `G`	[0] state alone	[0] shop

Key rules, read from `contracts/BunchState.cash`:

- **join.** The state is input 0 and holds the category's minting NFT; `n < C`; no other input may sit at `seatLB` or carry the bunch category (this blocks dragging a victim's seat into a join); state′ has the exact commitment with `count + 1` and `nextSeat + 1` and does not lose value; output 1 is at `seatLB` with value exactly `p` and commitment `53 || seatNo || buyerLB`, where `buyerLB` is the locking bytecode of output 2, the receipt `52 || seatNo`; at most four outputs, and output 3 carries no token.
- **complete.** `n ≥ T`; exactly `n + 1` inputs, every input after the first a real seat of this category; two outputs carrying no tokens, so every NFT burns.
- **refund.** `tx.time ≥ D`; `n < T`; exactly `n + 1` inputs and `n + 1` outputs; output `i` pays seat `i`'s stored address exactly what it is owed.

- **leave.** `n < T`; the receipt at input 2 is of this category and names the same seat number as the seat at input 1; no further bunch tokens; the seat's stored address is refunded at least the seat's value.
- **merchantAccept.** `1 ≤ n < T`; input 1 is the shop key (this category, commitment `4d`); exactly `n + 2` inputs; rebates appear in seat-input order.
- **reclaim / escape.** After `G`, a single input and a single output to the shop or to the seat's stored address.

Figure 2 summarises the life of a bunch.

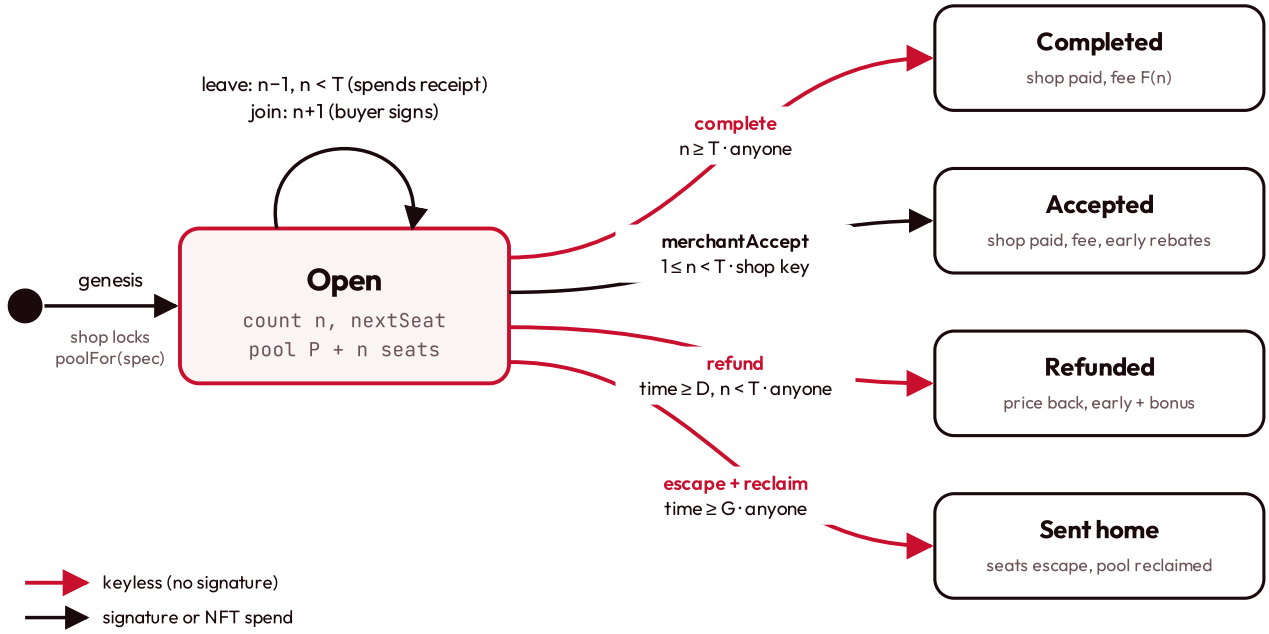


Figure 2. State transitions of a bunch. Red arrows are keyless paths that anyone may broadcast; dark arrows need a signature or an NFT spend.

4.6 Deadline semantics

CashScript checks time with `tx.time ≥ x`, which compiles to a locktime check: a transaction can prove that it is **at or after** a point, never before it. Consequences:

- **Height or time.** A locktime below 500,000,000 is a block height; above it, a Unix time compared against the median time past (MTP) of recent blocks. MTP runs about an hour behind the wall clock, so refunds on time deadlines open about an hour after the stated deadline. Demo bunches use block-height deadlines.
- **No "before".** The contract cannot block a join after the deadline. The UI hides Join at the deadline; a late seat is refunded in the same batch; if late joins fill the bunch it completes, which only helps buyers.
- **Grace period.** `G > D` bounds how long money can sit in an abandoned or broken bunch; the app limits `G - D` to 30 days.

4.7 Payout formulas

Let `P` be the value of the state UTXO (the pool) and `s_i` the seat number of seat input `i`. Let `E = |{ i : s_i < K }|` be the number of early seats present, so `E ≤ min(n, K)`.

Platform fee (`complete` and `merchantAccept`; integer division):

```
F(n) = max( floor(n · p · φ / 10000), 1000 )      fee output must equal F(n) exactly
```

complete (gross `Γ = P + Σ seat values = P + n·p`):

```
fee output  = F(n)
shop output ≥ Γ - F(n) - M
```

refund (bonus paid `B = E · b`):

```
buyer output i = p + b · [ s_i < K ]              exactly, to seat i's stored address
shop output    ≥ P - B - M
```

merchantAccept (gross includes the 1,000-sat key: `Γ = P + 1000 + n·p`):

```
fee output    = F(n)
rebate output = b, one per early seat              exactly, to that seat's stored address
shop output   ≥ Γ - F(n) - E·b - M
```

reclaim and **escape**: output `≥` input value `- M`.

Since every output the contract allows is either exact or flows to one fixed recipient with a `≥` lower bound, the only discretion left to a keyless broadcaster is the miner fee, bounded by `M`. This is Finding F1 (Section 7.3).

4.8 Sizing the pool and the fee budget

The shop must lock enough in the state UTXO that every refund path stays above dust after paying every possible bonus and the maximum miner fee:

```
poolFor(spec)  = K · b + M + 1000
maxFeeFor(C)   = 3000 + 300 · C        (≈ 1 kB + 200 B per seat at about 1 sat/byte, with
margin)
```

`maxFeeFor` reflects transaction size: BunchState's redeem script is about 777 bytes, so any transaction spending the state exceeds 1 kB, and each additional seat input adds to the size. Builders pay `size × 1 sat/byte + 50` sats. The app offers Join only when the state UTXO holds the minting NFT, the number of live seat UTXOs equals `count`, and the pool is at least `poolFor(spec)`.

4.9 Decision D1: rebates on a smaller deal

Because the contract cannot enforce "before the deadline", an earlier design that returned the whole pool to the shop on `merchantAccept` would let a shop accept after the deadline and skip the promised bonus. `merchantAccept` therefore pays `b` to every early seat present, as a rebate, and requires `n < T` (a full bunch completes instead). The promise to buyers becomes simple and always true: **you get the deal, or your money back; early joiners get the bonus unless the group fills.**

5. Economic analysis

5.1 Payoffs for a buyer

Consider a buyer who values the item at `v > p`. Let `c` be the miner fee the buyer pays to join (a join is about 1.5 kB; the app pays about 1 sat/byte), and ignore the 1,000-sat receipt output, which stays in the buyer's wallet. For a buyer holding seat number `s`:

Outcome	Early seat (`s < K`)	Late seat (`s ≥ K`)	Not joining
Group fills (`complete`)	`v - p - c`	`v - p - c`	0
Shop accepts smaller group	`v - p + b - c`	`v - p - c`	0

Outcome	Early seat (`s < K`)	Late seat (`s ≥ K`)	Not joining
Deadline passes (`refund`)	`b - c`	`-c`	0

Proposition. For an early seat, joining weakly dominates not joining when `v ≥ p + c` and `b ≥ c`, and strictly dominates it when both inequalities are strict, provided a failed bunch is settled by `refund` between `D` and `G`.

The last condition matters because, after `G`, anyone may `escape` a seat: it returns the payment minus up to `M` of miner fee and pays no bonus, and once one seat has escaped `refund` can no longer run (it needs every seat). Every open page broadcasts the refund as soon as the deadline passes, and the grace period is the window for doing so.

Each row in the early column is at least the corresponding entry in the "not joining" column under those two conditions. The condition `b ≥ c` matters: the app allows bonuses from 546 sats, and a join costs roughly the same order in miner fee, so a shop that wants strict dominance should set `b` comfortably above the join fee. The analysis also leaves out the time value of locked money, BCH price movement during the bunch, and the risk that the shop does not deliver after completion (Section 9).

For a late seat, joining is not dominant: the buyer loses `c` if the group fails. Joining is worthwhile when the probability of success `π` satisfies `π · (v - p) > c`. The live counter (Hu, Shi & Wu, 2013) and the early seats ahead of them raise `π` for late buyers, which is the intended cascade: the bonus recruits the first `K` buyers, and their visible commitment recruits the rest. A buyer may also `leave` before the target is reached and recover the seat in full, at the cost of another miner fee.

5.2 Who pays the bonus, and when

The bonus is paid by the shop, from the pool locked at genesis, in two cases only:

- **refund:** each early seat present receives `b`; the shop receives the remainder of the pool minus the miner fee.
- **merchantAccept:** each early seat present receives `b` as a rebate (D1).

On `complete` the whole pool returns to the shop with the sale proceeds, so a successful bunch costs the shop no bonus.

5.3 The shop's cost is bounded

Seat numbers are unique and never reused, so at most `K` early seats can exist. The shop's bonus outlay in any outcome is therefore

```
bonus outlay = E · b  ≤  K · b
```

and the shop also bears the miner fee of the settlement, at most `M`. Its capital commitment is `poolFor = K·b + M + 1000`, locked from genesis to settlement and returned in every outcome minus the bonus outlay and the settlement miner fee. If `π_f` is the probability that the bunch fails, the expected bonus cost is at most `π_f · K · b`. Tabarrok's own summary of the 2021 experiment adds that in equilibrium the bonus is never paid, because the bonus itself makes success the expected outcome; Bunch does not yet have field data to test this.

Two strategic concerns and the design's answers:

- **Bonus farming.** A buyer could take early seats in a bunch they expect to fail. The bonus is capped at 20% of the price, goes only to early seats, and seat numbers are never reused; if the shop accepts the group, the farmer becomes a paying customer.
- **Self-filling.** A shop could buy its own seats to complete and avoid the bonus. It then pays itself minus the platform fee and still owes goods to the real buyers; the UI labels seats whose refund address is the shop's.

5.4 Incentives of keyless settlers

Nobody is paid to settle. Settlement still happens because the interested parties run the page: buyers want their refund or their goods, and the shop wants its money. Settling costs the settler nothing, because the miner fee comes out of the bunch (from the shop's output on `complete`, from the pool on `refund`), capped by `M`. A malicious settler can at most waste up to `M` satoshis on miner fees; exact-output rules stop it paying anyone else (Section 7.3). Concurrent settlers race; the first broadcast wins and the others fail with a "missing inputs" error that the page ignores. After a rejected broadcast the page waits for the next block before trying again.

5.5 Fee model

The platform fee `F(n) = max(floor(n·p·ψ/10000), 1000)` is fixed in each bunch's contract at creation and is paid only when the bunch completes or the shop accepts a smaller group. A failed bunch pays no platform fee. The app sets `ψ = 200` (2%) when it is built with a fee address and 0 otherwise. The 1,000-sat floor keeps the fee output above dust but means small bunches pay a larger share: in the testnet3 rehearsal, two seats of 6,000 sats would owe 240 sats at 2%, and the contract required the 1,000-sat floor. On mainnet, two seats of 30,000 sats paid exactly 1,200 sats (Section 8).

6. Implementation

6.1 Stack

Layer	Technology (pinned versions from `docs/VERSIONS.md`)
Contracts	CashScript and `cashc` 0.14.0; `@bitauth/libauth` 3.1.0-next.8
Front end	React 19.3.1, Vite 7.3.8, Tailwind CSS 4.3.5; static files on Cloudflare Pages
Wallet	`@wizardconnect/core` 0.2.11, `dapp` 0.2.8, `react` 0.2.7 (`wallet` 0.2.5 in tests)
Chain data	Public Electrum / Fulcrum servers
Metadata	One stateless Cloudflare Pages Function serving BCMR registries
Tests	Node VM harness, Vitest 3.2.7, fast-check 4.10.3, Playwright 1.65.1

The core modules are small and separate: `spec.ts` (link codec and validation), `bunch.ts` (rebuild contracts and read state), `build.ts` (one builder per transaction type), `settle.ts` (keyless settler and outcome classification), `wallet.ts` (WizardConnect sign requests), `network.ts` (Electrum connection, subscriptions, broadcast), `bcmr.ts` (deterministic registry and genesis hash) and `voucher.ts` (pickup vouchers).

6.2 Signing with WizardConnect

WizardConnect pairs a web page with a phone wallet over encrypted Nostr relay messages, started from a QR code or deep link. The wallet announces chain-level extended public keys; the page derives receive and change addresses locally, scans them, and funds joins from **plain BCH UTXOs only**, filtering out token UTXOs and producing at most one change output, as the join covenant allows exactly three or four outputs.

Builders produce a CashScript transaction in which contract inputs carry their full unlocking bytecode and wallet inputs carry placeholder unlockers. The page sends a sign request with `broadcast: false` and `inputPaths` listing only the wallet's own inputs. The wallet signs those inputs and returns the transaction; the page broadcasts it on the network it is configured for. Broadcasting from the page, rather than from the wallet, guarantees that the transaction reaches the right network and lets the page show the txid at once. A join therefore needs one wallet approval.

Since WizardConnect has no message signing, authority for the two privileged actions comes from spending an NFT the user holds: the receipt for `leave`, the shop key for `merchantAccept`.

6.3 Reading state from Electrum

The page computes the BunchState and Seat addresses from the link, reads their UTXOs, and subscribes to both script hashes. Every open page sees each join as a subscription notification and redraws the counter. To find how a settled bunch ended, the page walks the state address history newest-first to the first transaction that is a real settlement and classifies it as completed, refunded or reclaimed; unrelated transactions (for example a dust send to the address) are skipped and unknown transactions leave the last good state on screen.

6.4 Settlement by any open page

The settler (`settleAction` in `src/core/settle.ts`) runs in every open page. If the state UTXO exists and the number of live seats equals `count`, it returns `complete` when `count ≥ T`, `refund` when the deadline has passed, and nothing otherwise. The page builds the keyless transaction, broadcasts it, and treats conflict errors as "someone else already settled".

6.5 Wallet metadata (BCMR) and vouchers

At genesis the shop's transaction commits, in an OP_RETURN output, to the SHA-256 hash of a Bitcoin Cash Metadata Registry (BCMR) document and a URI. The Pages Function `GET /bcmr/<category>.json` returns exactly those bytes, a pure function of the category, so wallets display seats, receipts and shop keys by name and icon (Figure 3). A snapshot test fixes the hash for a known category so that a copy edit cannot silently change what mainnet genesis transactions committed to. After completion a receipt doubles as a **pickup voucher**: the shop scans it and the page checks on chain that the receipt belongs to a completed bunch of that shop.

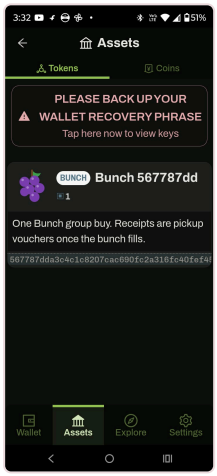


Figure 3. The receipt NFT from the mainnet join, shown in the Selene wallet with name, icon and description read from Bunch's BCMR registry.

7. Security analysis

Status: internal testing and internal review only. Bunch has not had an independent third-party audit. The contracts are per-bunch and immutable: a bunch always settles with the code that created it.

7.1 Threats and mitigations

✅ means a VM test proves the attack fails inside a contract.

Threat	Mitigation	Tested
Seat theft through a join or leave	Join and leave loops reject extra seat inputs and bunch tokens	✅

Threat	Mitigation	Tested
Early completion below target	`count ≥ target` and `inputs.length == count + 1`	✅
Refund before the deadline	`tx.time ≥ deadline`	✅
Refund redirected to an attacker	Output `i` must pay seat `i`'s stored locking bytecode	✅
Shorting an early seat's bonus	Output must equal `value + bonusEach` for `seatNo < earlyK`	✅
Join underpays or skips the counter	Seat value `== price`; exact next state commitment	✅
Leaving with someone else's seat	Receipt seat number must match the seat	✅
Escape or reclaim too early	`tx.time ≥ graceEnd`	✅
Fake state UTXO from another category	Every function checks the minting NFT of `bunchCategory`	✅
Forged receipt from another category	`leave` requires the receipt's category	✅ (found by mutation testing)
Fake seat parked at the seat address	Every seat loop checks the category	✅ (found by mutation testing)
Forged shop key	Key category must equal `bunchCategory`	✅ (found by mutation testing)
Refund that omits a seat but pays its slot to an attacker	`inputs.length == count + 1`	✅ (found by mutation testing)
Broadcaster shifts money between outputs	Exact fee, refund and rebate outputs (F1)	✅
Shop accepts after the deadline to skip the bonus	Early rebates on `merchantAccept` (D1)	✅
Fee griefing on keyless paths	Miner fee capped by `maxFee`; boundary tests	✅
Token smuggling or NFT leakage	No token on every payout and change output; fixed output counts	✅
Late joins after the deadline	Not enforceable on chain; UI hides Join	UI only
Shop non-delivery	Outside the covenant	Not mitigated
App disappears	`escape` and `reclaim` after `graceEnd`	✅

7.2 Methodology

Method	What it does	Result
VM attack tests	20 valid flows and 111 single-change attacks validated by CashScript's mock network provider on the real BCH VM. Each attack starts from a valid transaction and changes one thing; a rejection caused by dust, fee rate or locktime finality is flagged `BAD`, because such a test proves nothing	131 pass
Mutation testing	Each `require` becomes `require(true)` (rule mutants); comparisons, counters and branches are flipped (operator mutants: `==` → `≥`, `<` → `≤`, `≥` → `>`, `+1` → `+0`, doubled `maxFee`). The function's tests must fail	82/95 rule and 33/35 operator mutants killed; 0 unexplained survivors
Metamorphic tests	8 relations: state machine, join-order invariance, split invariance, refund conservation, leave then rejoin, target boundary, deadline boundary, no second settlement	Pass
Property tests	fast-check random lifecycles with invariants checked after every step	Pass (12 runs by default, configurable)
Builder tests	The app's own builders on the real VM, including 50-seat settlements	Pass (part of 56 Vitest tests)
Code review	An internal review pass over the app code; not a third-party audit	Fixes R3, R4, R6, R7

The 13 surviving rule mutants and 2 surviving operator mutants are each a rule that another rule already enforces. For example, `activeInputIndex == 0` in `join` survives because input 0 must already hold the only minting NFT of the category. Each reason is recorded in `MUTATION_REPORT.md`. The first run against the original build pack killed 72 of 91 rule mutants and exposed 8 missing negative tests (forged receipts, fake seats, a refund that orphans a seat). The contracts already blocked these attacks; tests now prove it.

The metamorphic relations follow Villanueva, Srinivasan and Rehman (2025), who applied metamorphic testing to a crowdfunding smart contract; per their abstract, the most effective relations killed 89% of mutants, against 25.65% overall.

7.3 Findings

ID	Finding	Our severity	Status
F1	Payout checks used `≥` with `maxFee` slack, so a keyless broadcaster could move up to `maxFee` sats between listed outputs (for example from the shop to the fee output). Nobody outside the listed outputs could receive anything	Low	Fixed: fee output `== F(n)`, each refund `== owed`; 1-sat overpay tests
D1	`merchantAccept` returned the whole pool to the shop, so a shop accepting after the deadline could skip the bonus	Medium	Fixed: rebates to early seats; requires `count < target`
M1	Mutation testing exposed 8 missing negative tests	Informational	Fixed: tests added
R1	A genesis built outside the app could start at `count = T - 1` with no seats and trap the next buyer's money	High (client)	Fixed in the app: Join only when live seats match `count` and the pool is at least `poolFor`
R2	A crafted link could set a very long grace period	Medium (client)	Fixed: grace period capped at 30 days
R3–R9	Voucher checks, settlement classification, dust sends, untrusted metadata, wallet timeouts, registry drift, settler retry rate	Low–Medium	Fixed

Severities are the project's own assessment.

7.4 Remaining risks

- **No "before" in CashScript.** Late joins cannot be blocked by the contract (Section 4.6).
- **One state thread.** Only one join can spend the state UTXO at a time; concurrent joiners conflict and the UI rebuilds and asks again.
- **MTP lag** on time deadlines, about an hour.
- **`maxFee` on keyless paths.** Up to `M` satoshis can go to miners instead of the shop (or, on `escape`, the buyer).
- **Fee floor.** The contract rounds the platform fee up to 1,000 sats even when `φ = 0`.
- **Client-side guards.** R1 and R2 protect users of this app; another client could still build a join into a malformed bunch.
- **Wallet behaviour.** Only Selene has signed on mainnet, and only a join. A wallet that alters contract inputs would fail to produce a valid transaction; it cannot redirect funds, because the contracts check every output.
- **Young dependencies.** The WizardConnect packages are at 0.2.x; all versions are pinned.
- **No third-party audit.** Use small amounts.

8. Evaluation and on-chain evidence

8.1 Automated tests

Layer	Count / scope	Status (Nov 22, 2026)
Contract VM tests	131 (20 flows + 111 attacks)	All pass
Mutation testing	95 rule + 35 operator mutants	82 + 33 killed; survivors explained
Vitest (builders, spec, metamorphic, property, wallet, demo)	56 tests	All pass
Live relay	Real WizardConnect dapp and wallet managers over a public relay	Passes when the relay is reachable
Browser journeys	4: complete with voucher check, accept with rebate, refund with bonus, leave	Pass
Smoke and accessibility	Deployed site on desktop and phone; axe WCAG 2 A/AA on every page	Pass

Continuous integration runs compile, typecheck, the VM tests, the Vitest suites and the build on every push. Mutation testing, browser journeys, smoke tests and the live relay test run locally, not in CI. No line-coverage figure is published.

8.2 Testnet3 rehearsal

The production builders were run with public test keys on testnet3, which follows mainnet consensus rules (`REHEARSE_NET=testnet3 npm run rehearse:chipnet`, Figure 4). Checked output values after broadcast:

Path	txid (prefix)	Observed
`complete` (keyless)	`080d8555…`	Shop paid; fee output exactly 1,000 sats (the floor; 2% of 12,000 is 240)
`leave` (receipt)	`495c311f…`	Buyer refunded exactly the 6,000-sat price
`refund` (keyless, after the deadline block)	`187d42c0…`	Buyer received exactly 7,000 sats (price + 1,000 bonus)
`merchantAccept` (`earlyK = 2`)	`f68bbc16…`	Shop 10,411; fee 1,000; rebate 1,000 to the early seat (D1)

In an earlier accept run with `earlyK = 1`, the only remaining seat was seat number 1 (the first buyer had left, and seat numbers are not reused), so the contract correctly paid no rebate. `escape` and `reclaim` have not run on any public network; they are covered by VM and builder tests only.

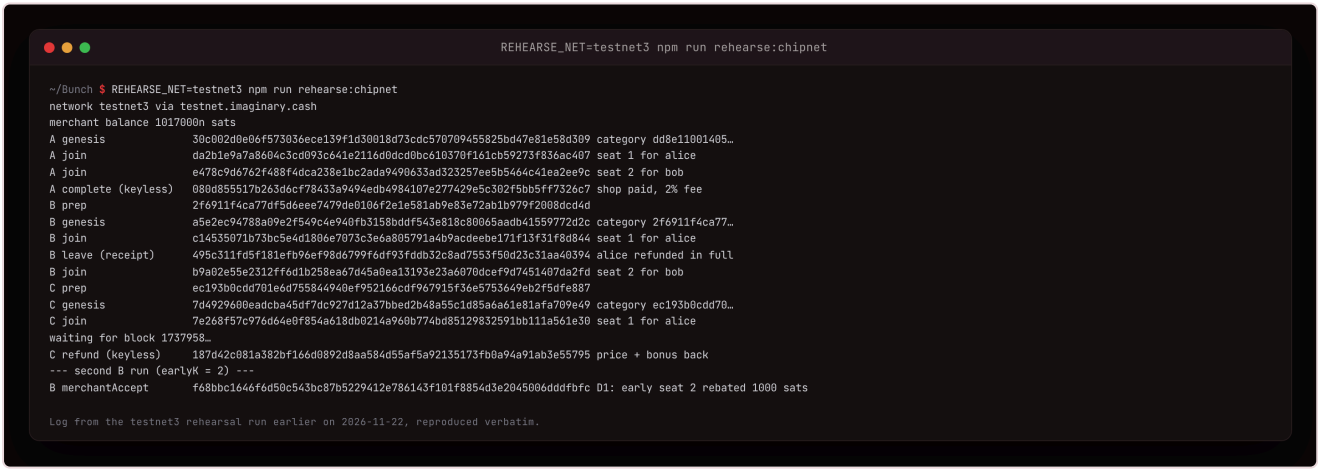


Figure 4. Log of the testnet3 rehearsal run of Nov 22, 2026: genesis, joins, keyless complete, leave by receipt, keyless refund with bonus, and merchantAccept with a D1 rebate.

8.3 Mainnet

On November 22, 2026 a real Selene phone wallet joined the "Judge Bunch" (two seats of 30,000 sats; seat 1 had been pre-filled by the team) over WizardConnect, and the open web page broadcast the keyless settlement. The phone wallet had been funded with 35,000 sats from the demo shop for the test, so this run demonstrates the mechanism with a real wallet; it is not evidence of an outside customer.

Step	txid	Result
Judge Bunch genesis	`5a00a53a7beb1eb76387351ba84825c4b59ad72db161e14c18cf1474b966e80a`	State thread with minting NFT and pool
Join (Selene, WizardConnect)	`34ddd25c3adf94dd7eaf1a03f964771f1221911bfd6ddc29a5a7e34347ffb609`	1,500 bytes; seat 30,000 sats; receipt NFT and change to the phone
Keyless complete (sent by the page)	`93ce7f3b964d28272a3b05f4ee8c6fd2f81fcfd8214d17b53cd37b2d0e9f0910`	1,399 bytes; shop 63,451 sats; fee exactly 1,200 sats

The completion can be reconciled from the transaction outputs (Figure 5). The state UTXO held a 6,100-sat pool and the two seats held 30,000 sats each:

```
gross        = 6,100 + 2 × 30,000                  = 66,100
fee  F(2)    = max(floor(2 × 30,000 × 200 / 10000), 1000) = 1,200     (2% of 60,000)
miner fee    = 1,399 bytes × 1 sat/byte + 50       = 1,449
shop output  = 66,100 - 1,200 - 1,449              = 63,451
```

Both state-spending transactions fall in the 1.4–1.5 kB range expected from the 777-byte redeem script.

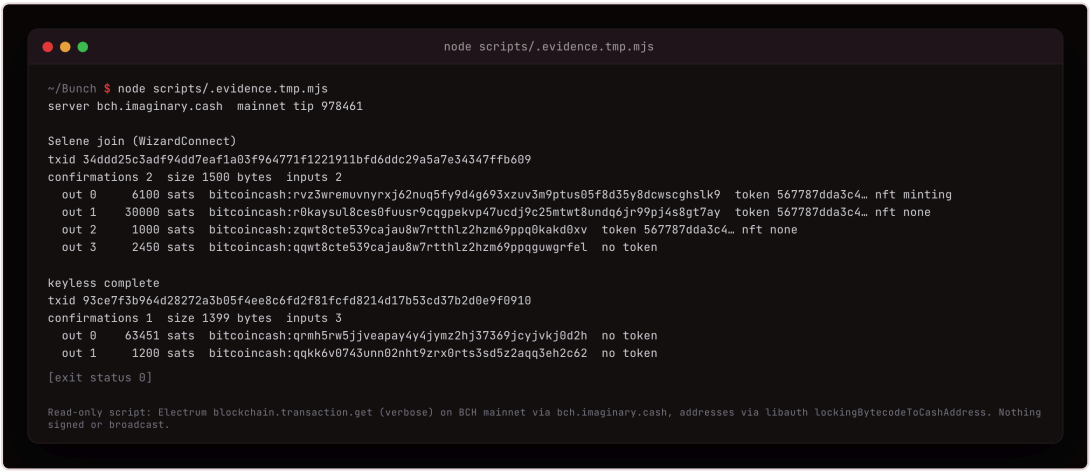


Figure 5. Both mainnet transactions read back from an Electrum server: sizes, every output value, address and token.

A second mainnet bunch (the "Refund Bunch", genesis `1925e921…`) needs three buyers but has only two seats, so it can never fill; after its deadline block every buyer is refunded and early seats receive the bonus. No mainnet refund transaction has been recorded yet.

9. Limitations

- **Thin on-chain coverage of signed paths.** A real phone wallet has signed only a join on mainnet. Create, leave and merchantAccept have been signed by headless WizardConnect wallets in tests and by script keys on testnet3, not by a phone wallet. Only Selene is recorded in the wallet matrix; Paytaca, Cashonize and Zapit are untested.
- `escape` **and** `reclaim` have never run on a public network.
- **Refund window.** The bonus is paid only if someone broadcasts `refund` between the deadline and `graceEnd`; after `graceEnd` a seat can be escaped without the bonus. The design relies on open pages to

settle in that window.

- **No audit, no users.** There has been no third-party audit, and there are no users, shops or revenue yet. The economic claims rest on laboratory studies, not on field data from Bunch.
- **Contention.** One state UTXO per bunch serialises joins.
- **Size limit.** The app caps a bunch at 50 seats so that settlement fits one standard transaction.
- **Price risk.** Bunches are priced in BCH; the fiat value of a seat can move during a bunch.
- **Delivery.** The covenant cannot force the shop to hand over the goods.
- **Wallet requirement.** Buyers need a BCH wallet that supports WizardConnect.
- **Regulation.** Pre-sale and consumer rules differ by country. Bunch never holds funds; the shop's terms belong on the listing.

10. Roadmap

All items below are **future work**; none is built. Contract changes ship with negative tests, a clean mutation report and a test-network rehearsal before mainnet.

Verification first. Phone wallets for every signed path (Selene, then Paytaca, Cashonize and Zapit); an external review of the two contracts before larger amounts are used.

Product gaps (no contract change). A leader flow ("start a bunch with friends", after Jing & Xie, 2011); public bunch history per shop as a trust signal; merchant tools (CSV export, "repeat this bunch", statistics); deadline reminders without a backend; more languages; an offline service worker.

Contract changes.

Item	Change	Main risk
A. Tiered prices	`complete` pays each seat back the difference when more buyers join (Anand & Aron, 2003)	Dust tiers must be rejected at creation
B. Stablecoin-priced bunches	Seats hold a stablecoin so fiat prices stay fixed	One token category per UTXO; design spike needed
C. Redeem-to-release	Money moves to a release covenant and the shop is paid per picked-up voucher	More steps for shops
D. Lanes	Several state threads per bunch remove join contention	`complete` must consume every lane and seat
E. XO template	XO invitations for join and refund	XO was pre-release when researched

11. Conclusion

Bunch shows that a dominant assurance contract can be enforced by a UTXO covenant rather than by a trusted organiser. The shop's promise to early buyers is a pool locked on chain; the rules for paying it are code that any participant can run; and settlement needs neither a server nor a key, because the covenant fixes who receives what and leaves the broadcaster no discretion beyond a capped miner fee. Bitcoin Cash's introspection opcodes and CashTokens make this possible with two small contracts, a link that carries the whole offer, and NFTs that stand in for message signatures. The implementation is tested at the transaction level, mutation-tested, and has settled real money on mainnet with exact payouts. What it lacks is equally clear: an external audit, more wallets, and real shops whose results would tell us whether the laboratory gains from refund bonuses carry over to group buying.

References

1. Tabarrok (1998). The private provision of public goods via dominant assurance contracts. *Public Choice*, 96, 345–362.
2. Cason & Zubrickas (2017). Enhancing fundraising with refund bonuses. *Games and Economic Behavior*, 101, 218–233. https://econpapers.repec.org/RePEc:eee:gamebe:v:101:y:2017:i:c:p:218-233
3. Cason & Zubrickas (2019). Donation-based crowdfunding with refund bonuses. *European Economic Review*, 119, 452–471. https://econpapers.repec.org/RePEc:eee:eecrev:v:119:y:2019:i:c:p:452-471
4. Cason, Tabarrok & Zubrickas (2021). Early refund bonuses increase successful crowdfunding. *Games and Economic Behavior*, 129, 78–95. https://econpapers.repec.org/RePEc:eee:gamebe:v:129:y:2021:i:c:p:78-95; working paper: https://business.purdue.edu/research/working-papers-series/2021/1326.pdf; summary: https://marginalrevolution.com/marginalrevolution/2021/06/better-crowdfunding.html
5. Cason, Tabarrok & Zubrickas (2025). Signaling quality. *Management Science*, 71(7), 5933–5947. https://econpapers.repec.org/RePEc:inm:ormnsc:v:71:y:2025:i:7:p:5933-5947
6. Anand & Aron (2003). Group buying on the web: A comparison of price-discovery mechanisms. *Management Science*, 49(11), 1546–1562. https://econpapers.repec.org/RePEc:inm:ormnsc:v:49:y:2003:i:11:p:1546-1562
7. Jing & Xie (2011). Group buying: A new mechanism for selling through social interactions. *Management Science*, 57(8), 1354–1372. https://econpapers.repec.org/RePEc:inm:ormnsc:v:57:y:2011:i:8:p:1354-1372
8. Hu, Shi & Wu (2013). Simultaneous vs. sequential group-buying mechanisms. *Management Science*, 59(12), 2805–2822. Press release: https://media.utoronto.ca/?p=4606
9. Villanueva, Srinivasan & Rehman (2025). Metamorphic testing for smart contract validation. arXiv 2501.09955. https://arxiv.org/abs/2501.09955
10. Modern Retail. How Pinduoduo is competing with Alibaba by building commerce communities. https://www.modernretail.co/platforms/how-pinduoduo-is-competing-with-alibaba-by-building-commerce-communities/
11. Y Combinator. Pinduoduo and the rise of social e-commerce. https://blog.ycombinator.com/pinduoduo-and-the-rise-of-social-e-commerce/
12. CashScript documentation. https://cashscript.org/docs/; WalletConnect guide: https://cashscript.org/docs/guides/walletconnect
13. WizardConnect documentation. https://docs.riftenlabs.com/wizardconnect/
14. BCH BLAZE 2026 hackathon brief. https://dorahacks.io/hackathon/bchblaze2026/detail
15. Bunch source code, contracts, tests and evidence. https://github.com/LSUDOKO/Bunch (`contracts/`, `MUTATION_REPORT.md`, `SECURITY.md`, `TESTING.md`, `docs/EVIDENCE.md`)

Appendix A. Contract function reference

Contract	Function	Caller	Preconditions	Outputs
BunchState	`join()`	buyer, one signature	state at input 0; `n < C`; no seat or bunch token among other inputs	state′ (`n+1`, `nextSeat+1`, value not reduced); seat `= p`; receipt `52‖seatNo`; ≤ 1 plain change
BunchState	`complete()`	anyone	`n ≥ T`; inputs = state + exactly `n` seats of this category	shop `≥ Γ - F(n) - M`; fee `= F(n)`; no tokens
BunchState	`refund()`	anyone	`tx.time ≥ D`; `n < T`; every seat; `n+1` outputs	buyer `i = p + b·[s_i<K]`; shop `≥ P - E·b - M`
BunchState	`leave()`	buyer, spends receipt	`n < T`; receipt seat number = seat's	state′ (`n-1`, `nextSeat` kept); refund `≥` seat value; ≤ 1 change
BunchState	`merchantAccept()`	shop, spends key	`1 ≤ n < T`; key `4d` of this category; every seat	shop `≥ Γ - F(n) - E·b - M`; fee `= F(n)`; rebates `= b`
BunchState	`reclaim()`	anyone	`tx.time ≥ G`; state alone	shop `≥ P - M`
Seat	`withState()`	anyone	input 0 holds the bunch's minting NFT	rules enforced by BunchState
Seat	`escape()`	anyone	`tx.time ≥ G`; this seat alone	stored refund address ≥ value `- M`; no tokens

Appendix B. Glossary

Term	Meaning
Assurance contract	Pledges are collected only if a threshold is met; otherwise they are returned
Dominant assurance contract	An assurance contract that also pays contributors a bonus if the threshold is missed
Bunch	One group-buying offer: a category, a state thread and its seats
BCMR	Bitcoin Cash Metadata Registry: JSON that wallets read to name and illustrate tokens
CashTokens	Native fungible and non-fungible tokens on BCH outputs
Commitment	The byte string carried by an NFT; Bunch stores state, seat and receipt data in it
Covenant	A script that restricts the outputs of the transaction that spends it
Early seat	A seat with seat number below `earlyK`; it earns the bonus on refund or accept
Electrum / Fulcrum	Indexing servers that report UTXOs and push address notifications
Introspection	Script opcodes that read the spending transaction's inputs and outputs
Keyless path	A contract function that needs no signature: `complete`, `refund`, `escape`, `reclaim`
`maxFee`	The largest miner fee a keyless broadcaster may take from the bunch
MTP	Median time past: the median timestamp of recent blocks, used for time locktimes
P2SH32	Pay-to-script-hash with a 32-byte hash, used for both covenants
Pool	BCH held in the state UTXO: bonuses plus the miner-fee budget plus 1,000 sats
Receipt	Immutable NFT `52‖seatNo` in the buyer's wallet; proves seat ownership and serves as pickup voucher
Shop key	Immutable NFT `4d` in the shop's wallet; authorises `merchantAccept`
State thread	The BunchState UTXO carrying the minting NFT whose commitment holds `count` and `nextSeat`
WizardConnect	Wallet-connection protocol over Nostr relays; signs transactions, not messages