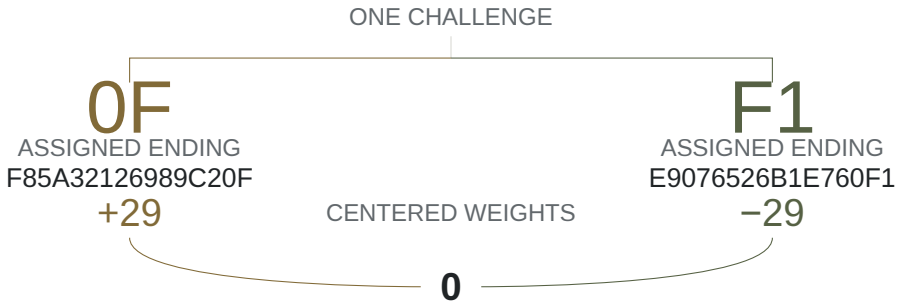
Bellpair

Arithmetic structure for verifiable work

Paired prime receipts in any number base

A form submission, an agent's tool call or a queued job can be cheap to generate and costly to handle repeatedly. A work receipt lets a service require a computational commitment to one specific action before accepting it. The requester searches; the service checks the result.

Bellpair provides a working implementation of that exchange. Its C/GMP solver produces two prime contributions, a separate verifier checks them, and an issuer accepts the completed receipt once. The contributions have assigned arithmetic roles derived from a number-theoretic reflection law. The MIT-licensed code includes the protocol, browser lab and reproducible measurements, so developers can adapt the exchange and test its costs.



A verified base-sixteen receipt. Both primes are shown in hexadecimal. Their endings select the reflected entries; the centered weights sum to zero.

The retained benchmark averaged about 957,000 candidate attempts per receipt. The verifier checks only the two returned contributions. Within the fixed 64-bit format, raising the work requirement does not make verification repeat the search.

A finite theorem supplies the arithmetic shortcut: a potentially enormous collision count reduces to a small calculation, and reflection assigns the matching roles. The lab makes that structure executable across bases 2–36.

Alexander S. Petty
Boston Agile Labs, established 2016

2026
Version 0.4

1 Work attached to an action

Make repeated actions carry a work requirement that another program can check.

Consider a public job queue. Receiving a job is easy; storing, validating and running it consume resources. A work policy can require the requester to return a fresh receipt before the job enters the queue. Repeated accepted jobs then need separately bound receipts, and a valid receipt can be checked without repeating the search that produced it.

Bellpair is a starting implementation for that policy. The issuer creates a challenge containing the message hash, arithmetic roles, difficulty and expiry. A worker finds both contributions. The issuer verifies the receipt and consumes its challenge once. Binding an application action requires a canonical encoding of the action and its content in that message.

Integration pattern. The shipped lab runs the worker and issuer locally.

Useful places to try it

Application	An experiment with a concrete outcome
Public submissions	Require a receipt for each form or submitted item. Measure ordinary-user delay and the cost of repeated submissions.
Agent tool calls	Bind the receipt to one requested action. Check expiry, retries and replay before the tool executes.
Job queues	Put a work requirement before enqueueing. Measure accepted throughput and verifier load as submission volume rises.

These are integration experiments a developer can build from the released components. The lab already provides issuance, solving, independent verification and one-use acceptance. A remote service must put the worker on the requester's side and use persistent, atomic redemption state. Work is one admission signal alongside authentication and rate limits; it can apply to automated clients without asking them to imitate a human.

Client puzzles have established uses in IKEv2 and Tor's onion-service defense [5, 6]. Bellpair contributes a particular arithmetic construction and the tools to evaluate it. Compare policies at matched solving cost, including slower devices, completion-time variation and server cost per accepted action. That comparison determines where to use it.

2 Finite structure at arbitrary scale

A count over a large residue system reduces to a finite address.

Take an integer $n \geq 2$ coprime to a base $b \geq 2$. Divide the nonzero residues modulo n into b digit bins. Multiply each residue by b, reduce it modulo n, and count how many residues remain in their original bin. This is a digit collision: agreement before and after a modular multiplication.

Writing $[x]_n$ for the representative in $\{1, \ldots, n-1\}$, the count is

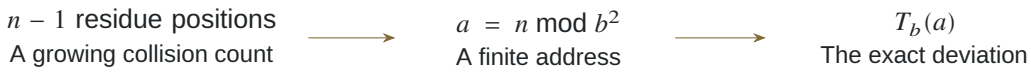
$$C_b(n) = \#\left\{1 \leq r < n : \left\lfloor \frac{br}{n} \right\rfloor = \left\lfloor \frac{b[br]_n}{n} \right\rfloor\right\}.$$

Its definition ranges over $n - 1$ positions. The finite determination theorem [1] removes that enumeration:

$$\boxed{C_b(n) = \left\lfloor \frac{n-1}{b} \right\rfloor + T_b(n \bmod b^2).}$$

The first term accounts for scale. The second is an exact integer from a table fixed by the base. Once the residue address is known, the table does not grow with n. It contains $\varphi(b^2)$ entries.

In base ten, 109 and 1009 both select the entry $+8$. Their collision counts are 18 and 108, respectively: a different scale term, the same deviation. The theorem applies to every coprime integer, including integers far beyond any feasible direct enumeration.



The base fixes the table before any prime is chosen.

Reflection adds a second piece of structure. The entries at a and $b^2 - a$ sum to -1; centering turns them into exact opposites. A verifier can therefore check both the fingerprint of an individual prime and its prescribed relationship to a partner.

This is the computational reduction Bellpair uses. It avoids reconstructing the collision process. Reading and reducing a larger integer still costs work, and primality has a separate cost. The finite theorem gives an exact shortcut for the collision statistic itself.

3 Reflection fixes the match

Every integer base has a collision table. Reflection fixes the balance of each pair.

The collision periodic table attaches an integer to each admissible ending. For base $b \geq 2$, put $q = b^2$, take $1 \leq a < q$ with $\gcd(a, b) = 1$, and set

$$G_b = \{d(b+1) : 0 \leq d \leq b-1\}, \qquad D_n(a) = \left\lfloor \frac{(n+1)a}{q} \right\rfloor - \left\lfloor \frac{na}{q} \right\rfloor.$$

The table entry is

$$T_b(a) = -1 - \left\lfloor \frac{a}{b} \right\rfloor + \sum_{n \in G_b} D_n(a). \tag{1}$$

This is the finite floor formula from the collision periodic table [1]. It can be evaluated by integer division.

Proposition 1 (Reflection)**.** *For every admissible a,*

$$T_b(a) + T_b(q-a) = -1.$$

Consequently $w_b(a) := 2T_b(a) + 1$ satisfies $w_b(q-a) = -w_b(a)$.

Proof. For $1 \leq n \leq q-2$, neither na/q nor $(n+1)a/q$ is an integer. The floor identities at reflected arguments give $D_n(a) + D_n(q-a) = 1$. At $n = 0$ the sum is zero; at $n = q-1$ it is two. Both endpoints belong to G_b, so the full increment sum is $(b-2) + 0 + 2 = b$. Also,

$$\left\lfloor \frac{a}{b} \right\rfloor + \left\lfloor \frac{q-a}{b} \right\rfloor = b - 1,$$

because a is not divisible by b. Substitution into (1) gives $-2 - (b-1) + b = -1$. Centering proves the second assertion. □

The prototype supports every integer base from 2 to 36. The mathematical law has no upper base limit. In base ten there are forty units modulo 100, giving twenty pairs $(a, 100-a)$ with $a < 50$. For example,

$$T_{10}(9) = 8, \quad T_{10}(91) = -9, \qquad w_{10}(9) = 17, \quad w_{10}(91) = -17.$$

The raw entries sum to minus one. The centered weights sum to zero. This distinction is part of the specification.

Reflection supplies the matching rule for the receipt. It holds throughout the admissible residue classes. Prime search and hash difficulty supply the work conditions; checking both assigned endings establishes the arithmetic balance. The balance follows from the roles and contributes no additional entropy.

4 Challenge-bound work

The challenge identifies the work. Every field below is included in the hash input.

The current protocol identifier is `reflection-work/v2`. A challenge contains six fields:

Field	Encoding and meaning
`id`	Fresh 16-byte identifier, as 32 lowercase hexadecimal characters.
`payloadHash`	SHA-256 of the message's UTF-8 bytes, as 64 lowercase hexadecimal characters.
`base`	Integer $2 \le b \le 36$; the modulus is $q = b^2$.
`pairA`	Admissible integer $0 < 2a < q$; the second role is $q - a$.
`difficultyBits`	Number of leading zero hash bits. The native verifier accepts 0 through 20; the local service issues 0 through 12.
`expiresAt`	Positive Unix timestamp in seconds, at most 9999999999.

The issuer rotates through the $\varphi(b^2)/2$ pairs for the selected base. This provides coverage of issued roles. Callers can abandon challenges, so completed receipts need not be uniformly distributed across pairs.

For each role $r \in \{a, q - a\}$, the worker searches a 64-bit nonce ν. The hash input is the following ASCII prefix followed immediately by the nonce's eight raw bytes in big-endian order:

```
reflection-work/v2
id=<32 lowercase hex characters>
payload=<64 lowercase hex characters>
base=<four decimal digits>
pair=<four decimal digits>,<four decimal digits>
bits=<two decimal digits>
expires=<ten decimal digits>
role=<four decimal digits>
nonce=
```

Every displayed line before `nonce=` ends with the single LF byte `0a`. There is no LF after `nonce=` or after the nonce. Decimal fields are zero-padded to the stated widths. The nonce is not hashed as printed hexadecimal text. These byte conventions prevent implementations from assigning different messages to the same displayed challenge.

SHA-256 is used as specified in the Secure Hash Standard [4]. Including the protocol, challenge, payload, base, pair, difficulty, expiry and role in the input binds the work to those fields under the usual hash assumptions. The issuer must still check that the challenge was actually issued.

Legacy version 1 retains its unchanged decimal encoding: no base field, two-digit pair and role fields, and domain `reflection-work/v1`. A supplied base always selects version 2, including base ten.

5 Two prime contributions

Each role must satisfy the hash target, assigned ending and prime test.

Let H be the 32-byte digest. Interpret its last eight bytes in big-endian order as u, and define the candidate by bitwise OR:

$$p = u \text{ OR } 2^{63} \text{ OR } 1.$$

Thus $2^{63} \le p < 2^{64}$ and p is odd. A contribution is admissible when:

1. the first `difficultyBits` bits of H are zero;
2. $p \bmod b^2 = r$, the assigned role;
3. p is prime.

The worker must find one contribution for each role. Distinct hash prefixes bind each contribution to its role; relabeling a contribution changes the message that the verifier hashes.

Primality is checked by trial division by the first twelve primes, followed by strong Miller–Rabin tests to the bases

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37.$$

Sorenson and Webster establish that the least composite passing all twelve bases is 318665857834031151167461, which exceeds 2^{64} [3]. Within the specified candidate range, passing this test therefore certifies primality. GMP performs the modular arithmetic.

OpenMP partitions the nonce range between workers. The first successful worker supplies the contribution, and the others stop. Scheduling can change which admissible nonce is returned and how many attempts are counted. Verification does not depend on that scheduling.

Recorded prime receipts

The original decimal walkthrough retains a real four-worker run at eight hash bits. This version-1 receipt remains independently checkable. Its two contributions are

$$\begin{aligned} p_1 &= 16845383891371498109, \quad & w_{10}(9) &= 17, \\ p_2 &= 11129527200732674791, \quad & w_{10}(91) &= -17. \end{aligned}$$

The run reported 577558 hash attempts, 50 primality tests and 0.513092 seconds of search. Both contributions were accepted by the separate C/GMP verifier. The full challenge, nonces and digests are retained in the downloadable receipt.

The current walkthrough also retains a separate version-2 receipt and verifier result for each of the 35 supported bases, at four hash bits and one worker. The display converts each prime to the selected base and highlights its final two digits. JSON prime strings stay decimal, so the wire representation is unambiguous.

Readers can reveal the recorded contributions and inspect their full receipts. A fresh local run issues a new challenge and records its own measurements.

6 Verification and one-use acceptance

Arithmetic validity and permission to use a receipt are separate checks.

A receipt carries the version, protocol identifier, complete challenge, two ordered contributions and the centered balance. Each contribution contains its role, a 16-character lowercase hexadecimal nonce, the prime as a decimal string, the digest as 64 lowercase hexadecimal characters, and the raw and centered collision values. The first contribution is for a; the second is for $q - a$. Primes are strings because 64-bit integers exceed the exact integer range of a JavaScript number.

The verifier validates the schema and domain before invoking the arithmetic checks. It recomputes each digest from the challenge and nonce, checks the hash target, derives the candidate, compares it with the supplied prime, checks the ending and proves primality within the 64-bit range. It then recomputes the two table entries and centered weights. It accepts arithmetic validity only if all checks pass.

The optional `search` object reports attempts, prime tests, elapsed seconds and workers. Those fields have no verification authority. They describe the producer's run; a valid receipt does not certify its claimed runtime or amount of hardware.

The separate executable checks the submitted proofs without repeating the search. An archived receipt remains arithmetically checkable after its challenge expires. That is distinct from permission to use it at a service.

Atomic acceptance

To redeem a receipt, the local issuer additionally requires:

1. a known challenge identifier;
2. equality with all six fields of the issued challenge;
3. a timestamp that has not expired;
4. successful independent arithmetic verification;
5. an unused challenge record.

Because verification is asynchronous, the service checks expiry and unused state again after verification. The final check and the transition to used state occur together in one event-loop turn. Two simultaneous redemption requests therefore cannot both succeed within that process.

This is a process-local guarantee. Records are held in memory. A restart forgets outstanding challenges, which are then rejected as unknown. A distributed deployment needs a persistent store with an atomic consume operation shared by every verifier instance.

7 Search and verification costs

These measurements isolate the cost of the additional filters at equal hash difficulty.

The retained benchmark uses the same message and nonce ordering for three search methods: hash difficulty alone, hash difficulty plus the selected ending, and both filters plus primality. It uses one worker, the $09/91$ pair, and four trials at each of 6, 8 and 10 hash bits. There are twelve pairs per method, thirty-six pair trials in total.

Required work	Mean attempts	Mean time (ms)
Hash difficulty	651.417	1.155
Hash and ending	59986.667	78.630
Hash, ending and prime	956517.833	1269.027

Repeated arithmetic verification averaged 0.031264 milliseconds over 1000 repetitions. The retained benchmark records compiler version 13.3.0 and GMP version 6.1.2. The times exclude process startup, HTTP handling and issuer state checks. They are measurements of this implementation and run, not portable performance guarantees. The archived benchmark does not retain a complete hardware description.

Equal hash difficulty makes the cost of additional filters visible. It does not compare schemes at equal total work. The paired-prime search performs more work here; an ordinary hash puzzle could also demand more work by raising its difficulty. The benchmark cannot distinguish a useful arithmetic advantage from that simpler change.

The reference implementation checks reflection throughout bases 2–36, retains the original decimal fixtures, and checks native solving and independent hash serialization in every supported base. Tests cover primality fixtures, canonical hash bytes, malformed receipts, altered proof fields, expiry and concurrent redemption. These are implementation checks. Their existence does not replace protocol analysis.

Local operating limits

The prototype restricts network access to loopback. It allows one solver job at a time, uses at most four workers by default, caps the search at 25 million attempts per role and stops it after 30 seconds. Challenges expire after ten minutes. Request bodies are limited to 16 KiB; the service limits stored challenge records and simultaneous verifiers. These are local resource controls, not a deployment-scale abuse analysis.

The public website presents a recorded walkthrough. Its production build has no active route to the solver. In local development the same interface can connect to the loopback service and exercise search, cancellation, verification, changed-digest rejection and replay rejection.

8 Base and work calibration

The arithmetic law stays the same. Table size and expected search work change.

In base b, there are $\varphi(b^2)$ eligible endings and half as many reflection pairs. Binary supplies one pair; hexadecimal supplies sixty-four. The table is a finite protocol parameter. Base ten has no privileged role.

Under a uniform-hash model and the usual prime-density heuristic near $X = 2^{64}$, the expected total attempts for two prescribed prime roles at d hash bits are approximately

$$\varphi(b^2) \log X \, 2^d.$$

This is a search-cost model, not a lower bound against optimized attackers. It explains why comparing bases at equal hash bits changes the amount of work demanded, as well as changing the arithmetic table.

A separate C/GMP audit used 96 pair trials per base per policy, one worker, and ten bases. One policy used four leading zero bits throughout. The other adjusted a 32-bit hash threshold using the displayed model to equalize expected attempts with base ten. The second policy calibrated expected work, not wall-clock time. Its hash domain was specific to the audit.

Base	Pairs	Equal-threshold attempts	Adjusted attempts	Adjusted time (ms)
2	1	1488	25639	34.28
3	3	4223	26323	35.08
4	4	6237	31067	41.05
5	10	15967	27897	32.96
6	6	9590	28198	32.38
8	16	24311	25552	29.94
10	20	29585	29585	34.60
12	24	30892	24520	28.42
16	64	102956	28955	33.69
30	120	167341	30509	35.06

All values are sample means. The run used an Intel Core i9-13900H, compiler 13.3.0 and GMP 6.1.2. Full trials retain medians, tails, prime-test counts and verification timings. Search-time dispersion and the small number of trials prevent these means from identifying a fastest base.

A smaller base gives a smaller table and fewer roles; that can suit a small implementation. A larger base gives more distinct reflection pairs. Power-of-two bases also allow bit-mask residue extraction, although this implementation has not measured an optimized advantage from it. Base choice should follow the needed role count and measured costs at matched work.

9 Bounded verification at scale

Search grows with the requested work. A receipt keeps verification bounded.

For a fixed base and the specified 64-bit candidate domain, verification always requires two hashes and two deterministic prime checks. It does not replay the nonce search or the previous receipt history. Processing R receipts takes $O(R)$ such checks. Every accepted pair contributes exactly zero centered weight, so arithmetic balance incurs no accumulated rounding or approximation defect as the receipt stream grows.

The saving persists for every finite receipt in an arbitrarily long stream. It comes from keeping each check local to its challenge. Raising the candidate bit width raises modular-arithmetic cost and requires a new primality guarantee. Raising the base grows the table to $\varphi(b^2)$ entries; direct evaluation of a fingerprint uses b integer-division terms. Precomputation trades that repeated work for storage. The prototype uses direct evaluation.

Independent nonce ranges can be searched concurrently. The reference worker uses OpenMP; a distributed worker would coordinate those ranges to avoid duplicated search. The verifier receives the two successful contributions regardless of how many workers participated.

A compact witness

Given the retained challenge, the two eight-byte nonces determine both hashes, candidates and weights. Thus a candidate compact witness needs 16 bytes, or 32 bytes with the 16-byte challenge identifier, before version and transport framing. The current interface deliberately uses a verbose JSON receipt for inspection; this compact transport is not implemented.

A follow-up component audit over all twenty decimal pairs measured about 57.36 microseconds for the full arithmetic check versus 4.16 microseconds for two hash checks on the same accepted inputs. That comparison isolates verifier cost. It does not compare admission policies at matched solving cost, and it excludes process, network and storage overhead.

Redemption state

Arithmetic verification can be stateless when supplied with the challenge. One-use acceptance cannot: the issuer must recognize and atomically consume an issued challenge. A deployed service needs shared, expiring redemption records, bounded verification concurrency and admission policies for different devices. These costs remain even when the collision balance is exact.

10 Building with arithmetic work

A reusable protocol today, with a mathematical route to richer completion rules.

Bellpair gives developers a complete exchange they can run, modify and measure. The worker produces a receipt; a separate program checks it; the issuer decides whether it may be used. Exact challenge bytes and retained examples make the agreement between those programs inspectable. That is useful engineering infrastructure for testing work policies.

For increasingly automated workflows, the design question is concrete. Which actions should require a computational commitment, and how much delay is acceptable before accepting them? A protocol that software can both produce and verify lets developers investigate that question directly. The current lab supplies the implementation and measurements needed to begin with one form, one tool call or one queue.

Richer arithmetic roles

Bellpair starts with the smallest collision fingerprint: agreement after one multiplication by the base. The broader finite determination theorem in *The Collision Invariant* [2] treats a shift of any positive length ℓ. For $n > b^{\ell+1}$ coprime to b, its collision deviation is determined by n mod $b^{\ell+1}$. That power of the base is necessary in general. Reflection continues to fix the paired sum at -1.

At each fixed depth, a finite residue address determines a collision statistic of an arbitrarily large integer. A finer address also determines the coarser addresses. This gives a specific direction for further work: completion rules built from several compatible arithmetic observations, with each required relation visible to the verifier.

Version 2 implements the one-shift pair. Extending it requires specifying the accepted combinations and measuring how the constraints interact. Those observations may be dependent; their number is not a measure of cryptographic strength. Prime-based work has prior art [7], and an advantage over conventional hash work needs a matched-cost study.

Start with a measurable decision

Choose one action, bind it to a fresh challenge, and measure the complete exchange. Include the requester’s search, verification, storage and one-use acceptance. Then vary the base and required work, and compare the result with the same action behind an ordinary hash puzzle.

The reason to try Bellpair is that this experiment is already within reach. The finite theorem supplies the arithmetic shortcut; the released code carries it through to a checkable receipt. Developers can begin with working components, inspect why an action was accepted, and learn whether this form of work is useful in their own system.

11 Evidence and references

The paper, recorded receipt and measurements are available from the Bellpair site.

The website distributes this PDF, a retained benchmark, the walkthrough receipts for every supported base, independent verifier results and the ten-base comparison. The matching TeX source is retained in the project repository. A manifest records artifact hashes and the source hash. The reference implementation is public under the MIT license at github.com/alexspetty/bellpair.

Within the reference repository, `make` builds the solver and verifier. The local service starts with `node server.mjs`. The receipt can be checked by passing its JSON path to `node verify-receipt.mjs`. The website's development view connects to that service on loopback port 8787. Its production view remains a public walkthrough.

Reproducible artifacts

Artifact	Contents
Technical paper	The arithmetic proof, challenge bytes, verification rules and scope.
Benchmarks	Original thirty-six pair trials and 1,920 cross-base trials.
Recorded receipts	One version-2 pair per base, plus the original decimal pair.
Verifier result	The independent arithmetic check for that recorded receipt.
Manifest	SHA-256 hashes for the distributed files and retained TeX source.

bostonagilelabs.com/whitepaper

References

[1] A. S. Petty, *The Collision Periodic Table*, research preprint, version history at doi:10.5281/zenodo.21853017.

[2] A. S. Petty, *The Collision Invariant*, research preprint, revised September 2026. doi:10.5281/zenodo.21855821.

[3] J. P. Sorenson and J. Webster, *Strong Pseudoprimes to Twelve Prime Bases*, Mathematics of Computation, 86 (2017), 985–1003. doi:10.1090/mcom/3134; arXiv:1509.00864.

[4] National Institute of Standards and Technology, *Secure Hash Standard*, FIPS PUB 180-4, August 2015. doi:10.6028/NIST.FIPS.180-4.

[5] Y. Nir and V. Smyslov, *Protecting Internet Key Exchange Protocol Version 2 (IKEv2) Implementations from Distributed Denial-of-Service Attacks*, RFC 8019, November 2016, section 4.4. `https://www.rfc-editor.org/rfc/rfc8019.html`.

[6] The Tor Project, *Introducing Proof-of-Work Defense for Onion Services*, 2023. `https://blog.torproject.org/introducing-proof-of-work-defense-for-onion-services/`.

[7] S. King, *Primecoin: Cryptocurrency with Prime Number Proof-of-Work*, 2013. `https://primecoin.io/primecoin-paper.pdf`.