Tare: An Optimizing Build for Anchor Programs, Proven by Differential Execution

Gross, tare and net compute for every instruction,
and an on-chain registry of the evidence

Tare
https://tare.cx

Version 1.0 · 22 November 2026 (UTC)

Abstract

Tare is an optimizing build tool for Solana programs written in Anchor and a public on-chain registry of what it built. It compiles an unchanged Anchor workspace twice, once with stock Anchor and once with a second implementation of the Anchor language substituted at the dependency level, and then rewrites the linked sBPF bytecode. Every instruction receives three weights in compute units: *gross* (the stock build), *net* (the Tare build) and *tare* (their difference). The two builds are shown to behave identically by executing them side by side on a local SVM over a corpus of transactions, drawn from the program's own tests, from sequences generated from its IDL and from samples of its mainnet traffic, and comparing account bytes, lamports, account metadata, logs, return data and errors after every transaction. The result is published as a record in a registry program on Solana mainnet, from which anyone can rebuild both binaries and rerun the comparison. Over a corpus of 189 open-source Anchor programs, 1,187 measured instructions fall from 6,295,676 to 4,459,739 compute units and binaries from 38,786,032 to 30,970,216 bytes, with 257,892 transactions matched and none diverging.

1 Introduction

1.1 Motivation

A Solana program pays for every instruction it executes. The runtime meters execution in compute units (CU), caps what one transaction may consume, and charges priority fees on the units a transaction requests [33]. A program's binary is stored on chain and locks rent in proportion to its size. Both quantities are properties of the compiled artifact, and for most programs the artifact is produced by one framework: Anchor [1].

Anchor gives a program its structure. A developer declares the accounts an instruction takes and the constraints they must satisfy, and the framework's macros generate the entry-point, the dispatch, the validation and the serialization around the handler. That generated code is where most of an instruction's compute goes. Section 11 measures it directly: across 1,187 instructions of open-source Anchor programs, the median instruction spends 46.8% of its compute units on work a second implementation of the same language does not need to do.

Teams whose programs press on compute limits have removed this cost by hand. The usual route is a rewrite on Pinocchio [4], a library of borrowed account views over the loader's

input buffer with no allocator. The rewrite of the SPL Token program in this style, p-token, was accompanied by an audit and a set of machine-checked equivalence proofs against the original [30]. The framework's own next major version, Anchor v2, is reached through a source migration [2]. In each case the program a team already wrote, tested and audited is replaced by a different program, and the team carries the cost of showing that the two behave alike.

Solar [10] makes the opposite choice for Solidity. It keeps the source language fixed and moves the optimizations developers write by hand into the compiler, arriving as a flag on the build tool they already use. Tare applies that structure to Solana, where the language being compiled is the framework itself.

1.2 Approach

Tare is a build tool. In an existing Anchor workspace, `tare build` replaces `anchor build`; no source file and no manifest changes. It produces two binaries from the same source. The *stock build* is what the pinned Anchor version produces. The *Tare build* compiles the same crates against `tare-lang`, a second implementation of the Anchor language that presents the same macros and types under the same package names and versions, and then passes the linked program through `tare-opt`, a rewriting stage over sBPF bytecode.

On Solana a program is upgraded in place over accounts that already exist, read by clients and indexers that already exist, so Tare requires the two builds to agree on everything those parties can observe: account bytes, lamports, account ownership and flags, logs, return data and errors. `tare prove` executes both builds on a local SVM over a corpus of transactions and compares all of these after each one. A single difference fails the proof and prints the input that caused it.

`tare attest` writes a record to a registry program on Solana mainnet holding the hashes of both builds, the source repository and commit, the hash of the corpus, the measured weights and the hash of the full report. `tare check` rebuilds from the record alone and reruns the corpus. `tare flag` files an input on which the two builds differ against a record, after confirming the difference locally.

1.3 Contributions

This paper describes the system as built at toolchain version 0.7.0 and measured on 2026-11-15.

- **Three weights.** A definition of gross, net and tare compute for an instruction and a program, measured by execution on identical inputs (Section 3).
- **Dependency-level substitution.** A build that swaps the implementation of a framework without touching the workspace, verifies the resolved dependency graph before and after compiling, and yields byte-reproducible artifacts (Section 5).
- **A second implementation of the Anchor language.** An entrypoint that borrows the loader's input buffer, direct discriminator dispatch, validation without reference counting, account access over the account's own bytes and allocation-free error paths, with the stock build's observable behavior (Section 6).
- **A bytecode stage for sBPF.** Relocation-aware rewriting and compaction of linked ELF programs, with a value-flow analysis that bounds the targets of indirect calls and rule tests that run on the production interpreter (Section 7).
- **A differential proof with a replayable corpus.** Side-by-side execution over three corpora, with the entire corpus packed into the report and committed to by one hash (Section 8).

- **An on-chain registry.** Records and flags as program-derived accounts, a fee on attestation, and the commands that write and verify them (Section 9).
- **An evaluation** over 189 programs from the Anchor repository and from production, with 257,892 transactions compared (Section 11).

1.4 Outline

Section 2 reviews how Solana executes a program and what a stock Anchor build runs around a handler. Section 3 defines the weights and the equivalence a proof establishes. Section 4 walks the pipeline end to end. Sections 5 to 8 cover the build, the two optimizing stages and the proof. Section 9 specifies the registry, and Section 10 the service that weighs deployed programs by address. Section 11 reports the measurements, Section 12 places the work, and Section 13 states the goal the project is built toward.

2 Background

2.1 Programs, accounts and compute

Solana separates code from state [41]. A *program* is an account whose data is an executable; everything a program reads or writes lives in other *accounts*, each with an address, an owner program, a lamport balance, a data buffer and a few flags. A transaction carries a list of instructions. Each instruction names a program, an ordered list of accounts with signer and writable flags, and an opaque byte string of instruction data.

Programs are compiled to sBPF, a register machine descended from the Berkeley Packet Filter [22] with eleven 64-bit registers, a fixed instruction width of eight bytes and a small set of syscalls into the runtime. The validator's virtual machine [5] verifies a program when it is deployed and charges one compute unit per executed instruction, plus fixed or size-dependent charges for syscalls such as logging, hashing, address derivation and cross-program invocation (CPI). A transaction that exhausts its compute budget fails. The budget is requested by the transaction, bounded per transaction and per block [33], and priority fees are priced per requested unit, so an instruction's compute cost decides both what it costs to send and what can be composed with it in one transaction.

Deployed programs are managed by the upgradeable loader. The program account holds a 36-byte pointer to a *ProgramData* account, whose data is a 45-byte header followed by the ELF file. An account must hold the rent-exempt minimum for its size, so a program of s bytes locks

$$R(s) = \mathit{rent}(s + 45) \tag{1}$$

lamports in its ProgramData account, where *rent* is the runtime's minimum balance function. An upgrade replaces the ELF bytes behind the same program address. Every account the program owns stays where it is, which is why a build that keeps account layouts unchanged can be deployed without a migration.

2.2 The loader's input buffer

When an instruction is invoked, the loader serializes its inputs into one contiguous buffer and passes its address to the program's entrypoint in register `r1`. Figure 1 shows the layout. The buffer begins with the number of accounts. Each account that has not appeared earlier in the list is written as a header of fixed size, its data, a reserve of 10,240 bytes into which the data may grow during the instruction, padding to an eight-byte boundary, and a final word. An account that repeats an earlier one is written as an eight-byte record whose first

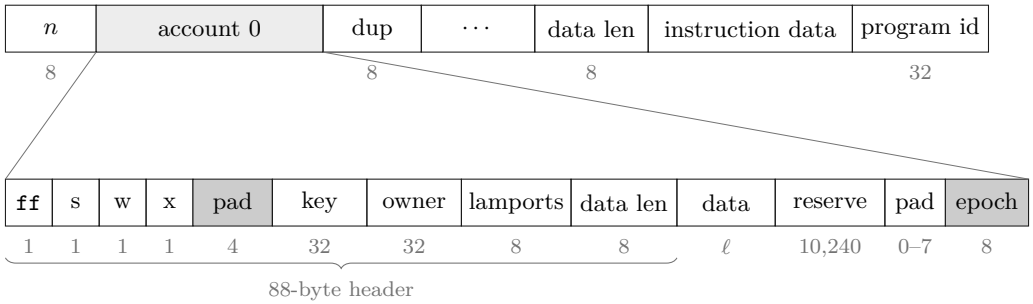


Figure 1: The input buffer the upgradeable loader passes to a program, with one non-duplicate account expanded. Sizes are in bytes; ℓ is the account's data length. `s`, `w` and `x` are the signer, writable and executable flags. The four padding bytes after the flags and the final word of each account (shaded) are not read by the runtime after serialization; Section 6 uses them.

```
#[event_cpi]
#[derive(Accounts)]
pub struct SetConfig<'info> {
    pub admin: Signer<'info>,
    #[account(mut, seeds = [b"config"], bump,
              has_one = admin @ RegistryError::Unauthorized)]
    pub config: Account<'info, Config>,
}

pub fn set_config(ctx: Context<SetConfig>, new_admin: Pubkey,
                  fee_recipient: Pubkey, attest_fee_lamports: u64) -> Result<()> {
    let config = &mut ctx.accounts.config;
    config.admin = new_admin;
    config.fee_recipient = fee_recipient;
    config.attest_fee_lamports = attest_fee_lamports;
    emit_cpi!(ConfigUpdated { admin: new_admin, fee_recipient, attest_fee_lamports });
    Ok(())
}
```

Listing 1: An instruction of an Anchor program: the account structure with its constraints and the handler. From `tare_registry`.

byte is the index of its first occurrence. The instruction data and the program's own address follow the last account.

After the program returns, the runtime reads lamports, owner, data length and data back out of the same buffer and enforces its own rules on them: only the owner may change data, lamports are conserved across the instruction, a read-only account is unchanged, and data may not grow beyond the reserve. These checks belong to the runtime and hold for any program.

2.3 What a stock Anchor build executes

An Anchor program is written as a module of handlers and a set of account structures. Listing 1 shows one instruction of the Tare registry, which is itself an ordinary Anchor program.

The handler is three assignments and an event. The macros expand the rest. In a stock build an invocation of `set_config` runs, in order:

1. **The entrypoint.** The SDK's entrypoint walks the whole input buffer and builds a heap vector with one `AccountInfo` per account. Each holds reference-counted, runtime-checked cells for the account's lamports and data.
2. **Dispatch.** The first eight bytes of the instruction data are the instruction's discriminator, the first eight bytes of SHA-256(`"global:set_config"`). Generated code matches it against every instruction of the program and logs `Instruction: SetConfig`.
3. **Arguments.** The remaining instruction data is deserialized with Borsh [8] into the handler's arguments.
4. **Account extraction and validation.** `try_accounts` consumes the account list in declaration order. For `config` it checks the owner, checks the eight-byte account discriminator, deserializes the whole account into an owned `Config` value, rederives the program-derived address from its seeds and compares it, checks the `has_one` relation and checks that the account is writable.
5. **The handler.**
6. **Exit.** Every account that could have been modified is serialized back into its data buffer in full.
7. **Errors.** On any failure an `Error` value carrying the code, the message, the account name and the compared values is built, formatted and logged before the program returns a status.

Steps 1, 2, 4, 6 and 7 are the same for every Anchor program. Their cost depends on the number of accounts passed and the size of the account structures, and not on what the handler does.

2.4 Where the weight is

Three properties of this expansion account for most of the compute a stock build spends outside its handlers.

Eagerness. The entrypoint materializes every account before the program knows which instruction was called or which accounts it will read. An instruction that receives ten accounts and reads two pays for ten.

Copying. An account is deserialized into an owned value and serialized back, even when the handler reads one field or changes one integer. For fixed-layout accounts the bytes in the input buffer already are the value.

Generality on the common path. Reference counting, heap allocation and formatted error construction serve cases that most instructions never reach, and the stock expansion pays for their machinery on the successful path as well: in the size of the values moved through each function boundary and in the code that surrounds every access.

Hand-written programs avoid all three. A second implementation of the framework can avoid them as well, provided it reproduces everything the first one makes observable.

3 The Three Weights

The name comes from weighing. A scale reads the gross weight of a container and its contents; the tare is the weight of the empty container; the net weight is what remains. Tare applies the three words to compute: the framework is the container, the program's own logic is the contents.

3.1 Definitions

Fix a program source P at a commit, an admitted Anchor version and a set of Cargo features. The build of Section 5 yields two ELF files, the stock build S and the Tare build B, each identified by the SHA-256 of its bytes. Let C be a *corpus*: a finite, ordered collection of transactions together with the account state each one starts from (Section 8).

Definition 1 (Sample)**.** Let ι be an instruction of P. An input $x \in C$ is a *sample* of ι when the transaction consists of a single top-level instruction addressed to P that selects ι, and the transaction succeeds on both S and B. The sample's values $u_S(x)$ and $u_B(x)$ are the compute units the runtime meters for that invocation on each build, including the cost of any programs it calls.

For a generated transaction the metered value is the transaction's consumed units; for a recorded one it is read from the runtime's log line for that invocation (Section 8.5).

Definition 2 (Gross, net and tare of an instruction)**.** Let $x_1, \ldots, x_k$ be the samples of ι in C, and for a build X let $u_X^{(1)} \le \cdots \le u_X^{(k)}$ be their values in ascending order. For $k \ge 1$,

$$G_\iota = u_S^{(\lfloor k/2 \rfloor + 1)}, \qquad N_\iota = u_B^{(\lfloor k/2 \rfloor + 1)}, \qquad T_\iota = G_\iota - N_\iota.$$

For $k = 0$ the instruction has no weights.

The statistic is the upper median: the middle value of an odd number of samples and the higher of the two middle values of an even number. It is always a value the meter reported for some execution and never an average of two. An instruction that no input reached successfully is reported as not measured and contributes nothing to any total.

Definition 3 (Weights of a program)**.** With I the set of instructions of P that have weights,

$$G_P = \sum_{\iota \in I} G_\iota, \qquad N_P = \sum_{\iota \in I} N_\iota, \qquad T_P = G_P - N_P.$$

The binary weights are the sizes $|S|$ and $|B|$ in bytes and the rent $R(|S|)$ and $R(|B|)$ of Equation 1.

A program total is a sum of per-instruction medians: it weighs one call of each measured instruction. The weigh service of Section 10 supplies call frequencies from mainnet for a deployed program.

3.2 Reading a weighing

Table 1 is the weighing `tare weigh` prints for `new_idl`, a program from the Anchor repository's test suite at version 0.32.3 with 13 instructions that exercise primitive, compound and empty account forms. The `empty` instruction takes no accounts and does nothing, so its gross weight is close to the fixed cost of entering an Anchor program and returning; its net weight is what the same entry costs in the Tare build. `generic`, `generic_custom_struct` and `no_case_conversion` create an account through a CPI to the System Program, whose cost is the same in both builds, so a larger part of their gross weight is net.

3.3 Observational equivalence

Tare states the equivalence of two builds over what a party outside the program can see.

instruction	gross	tare	net	inputs
`account_and_event_arg_and_field`	1,461	932	529	16
`empty`	204	58	146	12
`enm`	1,537	904	633	16
`external`	1,414	915	499	16
`external_non_anchor`	1,450	906	544	16
`full_path`	1,928	1,098	830	16
`generic`	6,741	2,451	4,290	24
`generic_custom_struct`	6,743	2,451	4,292	24
`no_case_conversion`	4,550	2,483	2,067	24
`primitive_types`	2,544	1,235	1,309	16
`strct`	2,130	989	1,141	16
`type_alias`	2,506	956	1,550	16
`unsized_types`	1,973	1,044	929	16
total	35,181	16,422	18,759	228

Table 1: The weighing of `new_idl` (Anchor 0.32.3, commit `0e115d4bd34b`), in compute units. The stock binary is 283,536 bytes and locks 1,441,241,720 lamports; the Tare binary is 249,768 bytes and locks 1,269,700,280. The proof compared 1,880 transactions. "inputs" is the number of samples, under the name the tool prints.

Definition 4 (Observation)**.** Let X be a build, σ an account state and t a transaction. Executing t on σ with X loaded at the program's address yields a post-state σ' and a transaction result. The *observation* $\mathsf{Obs}_X(\sigma, t)$ is the tuple of

1. for every tracked account, its data bytes, its lamports, and its owner, executable flag and rent epoch, or its absence;
2. the transaction's log lines in order, excluding lines of the exact form `Program <address> consumed <n> of <m> compute units`;
3. the return data and the address of the program that set it;
4. the transaction error, or none.

The tracked accounts are every account in the starting state and every account any instruction of the corpus names, other than the program itself. The excluded log lines are the runtime's own report of the quantity being changed. Every other line is compared, including the instruction names Anchor logs, messages a handler writes, event data and the text of errors.

Definition 5 (Equivalence over a corpus)**.** S and B are *equivalent over* C when, for every sequence in C and every transaction t in it, $\mathsf{Obs}_S(\sigma_S, t) = \mathsf{Obs}_B(\sigma_B, t)$, where σ_S and σ_B are the states each build has reached at that point of the sequence from the same starting state.

Each build carries its own state forward through a sequence, so a difference that first shows in state would also surface in every later transaction that reads it. A *proof* in this paper is a completed run of this comparison with no difference found. It is a statement about C, and a record commits to exactly which C by hash (Section 8.6).

The definition covers failure as closely as success. A transaction that Anchor rejects must be rejected by the Tare build with the same error code, the same logged message, the same named account and the same compared values, and when a transaction is wrong in several ways the same fault must be reported first.

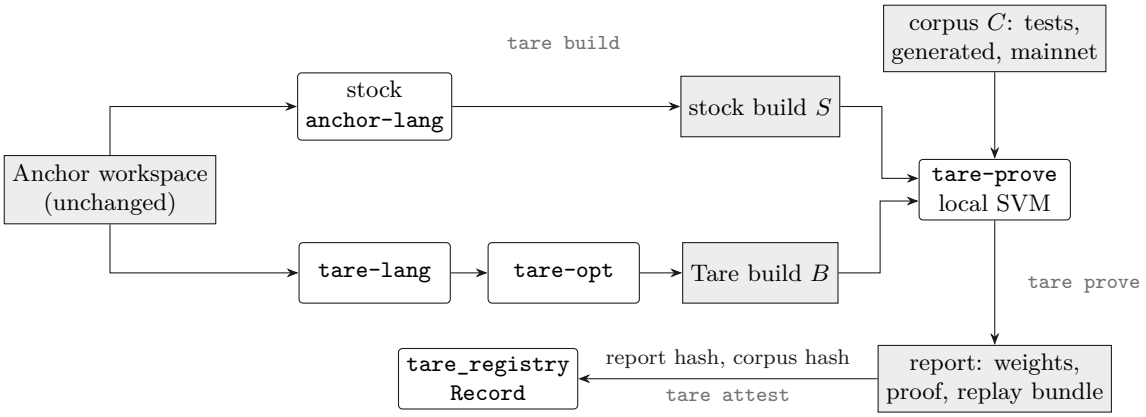


Figure 2: The pipeline. One workspace is compiled twice; both builds run over the same corpus on a local SVM; the report that results is stored by content hash and committed to by a record on chain.

4 System Overview

Tare has three parts: a toolchain that runs on the developer's machine, one program on Solana mainnet, and services that index that program and weigh deployed programs on request. Figure 2 shows the path from a workspace to a record.

4.1 Commands

The toolchain is one binary, `tare`, with six commands. Each takes `--manifest-path` to select a program in a multi-program workspace and `--features` to select Cargo features.

`tare build`
Compiles the workspace twice and runs the bytecode stage on the Tare build. It writes the stock build to `target/deploy/<program>.so`, the IDL to `target/idl/<program>.json` and the Tare build to `target/tare/<program>.so`, where `<program>` is the library name, and prints one line with both sizes.

`tare weigh`
Generates the corpus from the IDL, executes both builds over it and prints the weighing. The same data is written to `target/tare/weigh.json`.

`tare prove`
Performs the same execution and writes `target/tare/proof.json`. With `--program <address>` it first samples that program's recent mainnet transactions into the corpus. It exits zero only when every comparison matched.

`tare attest`
Uploads the report, then sends the registry's `attest` instruction signed by the team's keypair, and prints the record's address and its page.

`tare check <record>`
Reads a record and its report, fetches the recorded commit, rebuilds, reruns the corpus and prints `matches` or the first difference.

`tare flag <record> --input <file>`
Confirms that an input makes the two recorded builds diverge and, if it does, files it against the record.

Listing 2 is a session on `new_idl`, the program of Table 1.

```
$ tare build
building new-idl (Anchor 0.32.3)
bytecode: 8 instructions rewritten; 267880 -> 249768 bytes
new-idl: stock 283536 bytes -> tare 249768 bytes
$ tare weigh
new_idl                              gross       tare        net     inputs
account_and_event_arg_and_field       1461        932        529         16
empty                                  204         58        146         12
  ...
total                                35181      16422      18759        228
proof: matched (1880 transactions)
```

Listing 2: Building and weighing a program. Output captured from the tool; the instruction rows are those of Table 1.

4.2 Artifacts

The build's records are files under `target/tare/`. `build.json` records the toolchain version, the Anchor version, both artifact hashes, the source repository and commit, the selected features, the build environment and the account representation the build chose. `optimization.json` is the bytecode stage's report. `corpus.json` is the generated corpus and `replay.json` the complete replay bundle. `execution.json` holds the outcome of every transaction on both builds. `weigh.json` and `proof.json` follow the report schema shared with the registry's services. The two resolved lockfiles of the build are kept beside them with their hashes.

Each command verifies the files it consumes. Weighing refuses to run when either artifact's hash differs from the one in `build.json`, when the build was made by a different toolchain version, or when the selected features differ from the build's. Attesting has further preconditions, listed in Section 9.3.

4.3 The on-chain and off-chain parts

The registry program `tare_registry` holds three kinds of account: one `Config`, a `Record` per attestation and a `Flag` per filed input. Its mainnet addresses are in Section 9.7.

Reports are too large for an account, so a record stores the SHA-256 of its report and a location. The API at `https://api.tare.cx` stores reports and flag inputs by content hash, serves registry reads from a database an indexer fills from the program's events, and runs the weigh service. The site at `https://tare.cx` is read-only and connects no wallet: attesting and flagging are signed from the command line, as a deploy is.

5 The Build

The build reads the workspace and never writes it. Source files and manifests are compiled in place. The one file Cargo itself rewrites, `Cargo.lock`, is restored to its original bytes, or removed again if it did not exist, when the build exits. After compiling, the build compares the program's source file and manifest with the bytes it read at the start and fails if either changed.

5.1 Admission

`tare build` asks Cargo for the workspace's metadata, selects the package that builds a program library, and determines the Anchor version it resolves to: from the dependency's

own manifest when Anchor is a path dependency, from `Cargo.lock` otherwise. Four version lines are admitted: 0.31.1, 0.32.1, 0.32.3 and 1.2.2. Any other version stops the build with the version, the manifest and the line that declares it. A dependency graph that contains two versions of one Anchor package is refused in the same way.

The same holds for any construct Tare does not implement: the build stops and names the construct and its location.

5.2 Substitution by dependency patch

The Tare build compiles the same crates as the stock build against a different implementation of the framework. Cargo supports this directly: a `[patch]` table replaces a dependency's source with another package of the same name and a compatible version [35]. Tare passes the patch on the command line as a configuration file, so no manifest in the workspace mentions it.

The patch has to match whatever the workspace depends on. Tare first resolves the stock graph and reads from it every Anchor framework package the program uses and where each comes from. For each one it generates an *adapter*: a package whose name and version are exactly those of the Anchor package and whose contents are Tare's implementation for that version line. The configuration file then carries one patch table per source kind: `[patch.crates-io]` for registry dependencies, `[patch."<git URL>"]` for git dependencies, and a patch keyed by a `file://` URL for path dependencies, which is how the programs inside the Anchor repository itself depend on the framework. Dependencies between Anchor packages point to the matching adapters. Packages that only describe the IDL schema keep their upstream definitions; everything that generates or runs on-chain code comes from Tare.

Each adapter directory is named with a digest of the resolved dependency graph, so one program's selection of Solana crates cannot alter another program's adapter.

5.3 Verifying the graph

A patch that silently fails to apply would produce a second stock build under Tare's name. The build therefore checks the graph that Cargo actually resolved, before compiling and again after, and guards the compilation in between.

Before compiling, it runs `cargo metadata` with the patch and requires every Anchor framework package in the resolved graph to point into the adapter root. A single stock framework package fails the build. The framework allowlist is explicit, so an unrelated user package whose name happens to begin with `anchor-` is not affected.

During compilation, the compiler wrapper described below refuses to compile an Anchor framework crate from any location outside the adapter root. `cargo-build-sbf` runs its own preliminary `cargo metadata`; Tare supplies itself as the Cargo executable for that call so that the same patch and lockfile apply to it.

After compiling, it resolves the graph once more with `--locked`, verifies it again, and requires the lockfile on disk to be byte-identical to the one verified before compilation.

Both resolved lockfiles, stock and Tare, are stored under `target/tare/` with their SHA-256 hashes, and both resolved graphs are stored as JSON.

5.4 Two compilations under the same settings

Both builds use the same platform tools, the same target architecture (`--arch v0`), the same profile and the same workspace settings. They differ in the dependency patch and in nothing else the developer controls. The Tare build is compiled first; the stock lockfile is then reinstated and the stock build compiled. Each Anchor version line has its own compilation

caches for the stock build, the Tare build and the IDL build, and a file lock per version line covers the whole sequence from compilation to the copy of the final ELF, so concurrent builds of same-named programs in different workspaces cannot exchange artifacts.

The IDL is produced by Anchor's own compile-time IDL generation from the unchanged source. It belongs to the stock side entirely. Where a production source uses custom serialization that the stock IDL compiler cannot describe, `--idl` accepts an explicit IDL file; the build validates it against the IDL schema, announces that a supplied IDL is in use, and pins the file's content inside the replay bundle.

After both ELF files exist, each is admitted through the validator's own loader. `tare-opt` loads it with the Agave runtime [3] that LiteSVM [17] embeds, and a separate checker pinned to Agave 4.4.0, a later release than the one LiteSVM embeds, runs the deployment, verification and JIT compilation paths on both files. A build whose Tare artifact fails admission deletes the artifact and reports the loader's diagnostics.

5.5 Reproducible artifacts

A record is checked by rebuilding, so the same source must produce the same bytes on another machine in another directory. A build is reproducible when the same source, environment and instructions yield bit-identical output [29, 13]; the usual obstacle in Rust is that absolute paths enter the artifact through panic messages, debug information and the metadata hash that disambiguates crates.

Both compilations run through one compiler wrapper. It leaves Cargo's optimization and profile flags as they are and changes two things. It remaps the source root, the Cargo home, the Rustup home and the generated adapter directories to fixed prefixes. And it replaces the crate-disambiguation metadata Cargo derives from absolute dependency locations with a stable identity computed from the package name and version, its normalized source location, the enabled configuration flags, the target and the crate kind. Procedural-macro crates built for the host receive the same treatment as crates built for sBPF. `build.json` records the scheme as `remap-metadata-v1`.

With this in place, two checkouts of one commit in different directories with empty caches produce byte-identical stock and Tare artifacts. Because the stock artifact is reproducible too, a record's stock hash can be verified by anyone, and the pair of hashes identifies the comparison exactly.

5.6 The build environment

By default every command runs inside a container image. The CLI mounts the repository root at `/workspace`, sets the working directory to the program's position inside it, and invokes the same binary in the image with `--host`. The image reference defaults to `tare-toolchain:<version>` and is selected with `TARE_IMAGE`. A lexical mapper converts host paths, including drive letters and a source root above the working directory, to POSIX paths inside the container without consulting the host's path rules. With `--host` the commands run directly on the machine.

The environment is part of the result: `build.json` and every report record either the image reference or the host's target triple. `tare check` compares the recorded environment with its own before rebuilding and, when they differ, names both.

5.7 Choosing the account representation

Unchanged source can depend on more than the documented behavior of the framework's types. Some programs pass `AccountInfo` values to SDK functions that require the SDK's

concrete type; a few inspect the type's layout, its pointers or its reference counts. Tare's fastest account type is a compact view over the input buffer, which is a different concrete type from the SDK's. The build decides per program which representation the Tare build uses, and records the decision and its reason.

1. The source is parsed and searched for *representation observations*: layout queries, pointer or reference-count inspection of account types. A query over a locally defined payload type whose fields can be checked recursively does not count. If an observation is found, the program is compiled with SDK-compatible accounts.
2. Otherwise the program is compiled against the raw representation. If it type-checks, that build is used. This is the case for 180 of the 189 programs in the benchmark.
3. If compilation fails with a type error that names an account type, the program has a typed SDK boundary. It is compiled again with compact SDK-backed views: accounts are still parsed lazily from the input buffer, and a real SDK object is materialized once for each account that crosses the boundary and reused.

All three branches run Tare's validation and Tare's runtime. The choice affects how an account is represented in memory and never which checks are performed.

One further gate applies to every build: sBPF limits a stack frame to 4,096 bytes, and the compiler reports frames that exceed it as a diagnostic while still emitting a file. Tare treats that diagnostic as an error and does not admit the artifact.

6 The Front Stage: A Second Implementation of the Anchor Language

`tare-lang` is the crate the adapters of Section 5.2 present as `anchor-lang`, with `tare-lang-macros` behind its attribute and derive macros and `tare-spl` as `anchor-spl`. It implements the public surface an Anchor program is written against: `#[program]`, `#[derive(Accounts)]` and its constraints, `#[account]`, `#[event]`, `#[error_code]`, the account types (`Account`, `Signer`, `Program`, `SystemAccount`, `UncheckedAccount`, `AccountLoader`, `Interface`, `InterfaceAccount`, `Sysvar`, optional and boxed forms), `Context` and `CpiContext`, the `require!` family, `emit!` and `emit_cpi!`, and `declare_program!`. A program compiled against it contains none of the stock framework's runtime code.

The implementation follows the pinned Anchor source for everything observable (the order in which accounts are extracted and constraints checked, the error raised by each failed check, the bytes written to accounts and logs) and Pinocchio's approach [4] for everything that is not: how the input is parsed and how accounts are represented in memory. Table 2 sets the two builds side by side over the steps of Section 2.3.

6.1 The entrypoint

The generated entrypoint receives the address of the input buffer of Figure 1 and scans it once. The scan has to reach the end of the account list because the instruction data and the program's address come after it. For each non-duplicate account it performs two writes into bytes the runtime does not read back:

- the account's data length at entry is copied into the four padding bytes of the header, so that later growth can be checked against the 10,240-byte reserve without separate bookkeeping;

Step	Stock build	Tare build
Entry	Every account parsed into a heap vector of reference-counted descriptors	One scan of the input buffer; descriptors built on demand from the buffer itself
Dispatch	Discriminator matched, then a shared path into the handler's wrapper	Discriminator matched directly to a per-handler wrapper that returns the status word
Arguments	Borsh deserialization	Fixed-width reads for integer, boolean and address arguments; Borsh for the rest
Accounts	Owned value deserialized from each typed account	A view over the account's bytes where the layout allows; a snapshot otherwise
Exit	Each mutable account serialized in full	Views need no write; snapshots are written only if touched
Errors	Heap-formatted error value moved through each result	Static descriptors by reference, stack formatting, one shared logging boundary

Table 2: What each build executes around a handler.

- the final word of the account, where the loader serialized the deprecated rent epoch, is set to zero. It becomes the account's borrow state.

The scan also records whether any duplicate record is present. It allocates nothing and builds no descriptor. What it hands to dispatch is a two-word *account input*: a pointer to the first account and the duplicate flag. A cursor over the accounts is constructed from it only after an instruction has been selected.

Advancing the cursor past a non-duplicate account adds the header size, the original data length and the reserve, rounds up to eight bytes and adds the final word; past a duplicate it adds eight. Taking the next account yields a small descriptor: references to the key and owner inside the buffer, the three flags, and a pointer to the account header from which lamports and data are reached. When the record is a duplicate, the cursor walks from the first account to the indexed one, so both positions resolve to the same header and therefore to the same lamports, data and borrow state.

6.2 Borrow state in the buffer

Anchor programs rely on the SDK's runtime borrow checking: two live mutable borrows of one account's data are an error the program can observe. The stock representation implements this with a reference-counted cell per account for data and another for lamports. Tare keeps the same discipline in the word the entrypoint cleared: two signed 32-bit counters, one for data and one for lamports.

A shared borrow fails if the counter is negative or saturated and otherwise increments it. An exclusive borrow fails unless the counter is zero and sets it to -1. Releasing a guard restores the counter. A failed borrow returns the SDK's `AccountBorrowFailed`. Where a single check covers both resources before a write, the two counters are read as one 64-bit word and compared with zero.

Because the counters live with the account and not with a descriptor, every descriptor for an account, including those reached through a duplicate record and those exposed to SDK code, sees the same state. A CPI validates outstanding borrows before invoking the runtime, as the SDK's invocation does. Resizing an account checks the reserve against the stored original length, refuses when the data is borrowed, zero-fills growth, and writes the new length to the header where the runtime will read it.

6.3 Dispatch

The `#[program]` macro emits a match on the eight-byte discriminator that jumps to a wrapper generated for each handler. The wrapper performs, in the stock order, the declared-program-id check, the log of the instruction's name, argument decoding, account extraction, constraint checks, the handler call, the return-data write and the exit of each account. The instruction-name log uses a constant byte string. A wrapper returns the runtime's status word; a failed `Result` from the public Anchor interface is logged once at a shared boundary and lowered to that word, so the successful path does not carry a large result value through each call.

Arguments of fixed width (integers, booleans and addresses) are read directly from the instruction data after the same length check and the same boolean-validity check Borsh applies. Floating-point arguments, variable-length arguments and any argument whose type name the program shadows keep their Borsh decoder.

Anchor's built-in instructions are dispatched separately from the program's own. For the 0.x version lines these are the on-chain IDL operations (create, create buffer, write, set buffer, set authority, resize and close), which Tare implements with the stock behavior; for 1.x they are present when the program opts in. The event-CPI entry used by `emit_cpi!` is implemented for all lines.

6.4 Account extraction and validation

The `Accounts` derive generates the same sequence of extractions and checks as upstream, in the same order, because the order decides which error a faulty transaction reports. Generated validation writes the finished accounts structure into storage supplied by the caller, and only after every check has passed; the stock expansion returns the structure inside a `Result` and moves it. Loading a typed account goes through one out-of-line function per wrapper and payload type that writes into the caller's storage. Optional accounts keep the stock sentinel and the stock order of missing-account errors, and boxed accounts allocate only after validation succeeds.

Constraint generation covers initialization with payer and space, `init_if_needed`, seeds and bumps, ownership, fixed addresses, executable and rent-exemption checks, `has_one` relations, custom predicates with custom errors, `zero`, `close` and `realloc`, and the token, mint and associated-token constraints of `anchor-spl`. Where the version lines differ, each is followed: the 1.x reallocation path checks a nonzero resize against the same payer and destination before later constraints and reports the stock error with the account name and compared keys, and 0.x keeps its own order.

`Context.remaining_accounts` has the stock slice interface. The wrapper materializes the remaining descriptors only when the handler can observe them; zero or one stays on the stack.

6.5 Account data: views and snapshots

A typed account `Account<T>` holds its value in one of two forms.

A *direct view* is a pointer to the account's bytes in the input buffer, typed as `T`. Reading a field is a load from the buffer and writing a field is a store into it, so there is nothing to deserialize on entry and nothing to serialize on exit. This is sound only when the in-memory layout of `T` coincides with its Borsh encoding. The `#[account]` macro decides this per type: every field must be a fixed-width primitive (integers up to 128 bits, booleans, addresses, and fixed arrays of these), and the generated code checks that each Rust field offset equals the field's offset in the Borsh encoding, that each field's type is the primitive its name says and

not a shadowing definition, and that the type needs no destructor. A type that fails any of these uses a snapshot. The loader aligns account data to eight bytes; a type that asks for more is checked against the actual address, and wide fields read through the fixed-width decoder use unaligned loads. The view is requested only for an account the instruction declares mutable, in a handler whose typed accounts are known to be distinct: when the input contains duplicate records, the cursor checks that the typed prefix of the account list has none before a view is taken.

A *snapshot* is an owned `T` deserialized with Borsh, as in the stock build, with a dirty flag. Dereferencing mutably, replacing the value, reloading, or handing the account to code that can write to its bytes sets the flag. On exit a snapshot that is clean is not serialized. A snapshot of an account that may alias another is treated as dirty from the start.

For both forms an account being initialized has its discriminator written at exit, as stock Anchor does. Under 1.x, an account whose ownership moved away from the program during the instruction is handled as stock handles it: the value is serialized into a comparing writer that writes nothing, and a difference from the bytes in place raises the stock ownership error with the stock compared keys.

`AccountLoader` for zero-copy types exposes the same checked borrow guards over the buffer in both builds.

6.6 Errors

An Anchor error is observable in three places: the numeric code the transaction fails with, the log line carrying the error's name, number and message, and for many checks further lines carrying the account name and the two values compared. All three are part of Definition 4.

Tare's error value stores its static parts (name, message, origin) as references to constants and its compared values in compact inline storage, falling back to an owned allocation only when a value does not fit. The value occupies 144 bytes where the stock one occupies 264, so less is moved across every function boundary an error could cross. Messages are formatted into a 1,024-byte buffer on the stack; a message that does not fit is formatted on the heap in full, so no text is truncated. Addresses in error lines are rendered by a bounded base58 encoder that works on the stack.

6.7 Cross-program invocation, events and SPL

`CpiContext`, the `cpi` clients generated for a program's own instructions and the bindings `declare_program!` generates from an IDL produce the same instruction bytes and the same account order as stock. System Program calls made during account initialization marshal their account descriptors and signer seeds on the stack through the runtime's C ABI. CPI return values keep each version line's semantics: 1.x captures the data and the program that set it, earlier lines read it on access.

`emit!` logs the same data line. `emit_cpi!` performs the same self-invocation with the same event-authority account, so indexers that read events from inner instructions see identical bytes.

`tare-spl` provides the token, token-2022, token-interface, associated-token, token-extension, metadata and memo wrappers and the matching account types. Its wrappers perform CPIs to the deployed SPL programs. During a proof the same SPL program binaries are loaded on both sides.

Rent-exemption thresholds are computed with an exact integer path where it is equivalent to the stock floating-point computation and with the general computation outside that range.

6.8 What stays identical

The properties a deployed program's surroundings depend on are fixed by construction or checked by the proof:

Property	How it is held
IDL	one IDL, generated by stock Anchor from the unchanged source
Instruction discriminators	derived from the same names; exercised by every corpus transaction
Account layouts and discriminators	layout checks in generated code; account bytes compared
Error codes, names and messages	compared as transaction errors and log lines
Events	compared as log lines and as inner-instruction effects
Program logs	compared line by line
Return data	compared with its program address
Lamport movements	compared per account

7 The Bytecode Stage

`tare-opt` takes the linked Tare program, an ELF file of sBPF version 0, and returns a smaller ELF file with the same behavior together with a report of what it changed. It works below the level of any framework: its input is bytes, and nothing in it knows about Anchor.

In a linked program the addresses are already fixed, and they are referenced from places a rewriter must find: branch and call displacements, relocation entries, symbol tables, the dynamic section, the ELF entry point, and pointers stored in read-only data. A rewrite is admissible only if every such reference still denotes what it denoted before.

7.1 Admission and structure

The stage accepts a little-endian 64-bit ELF of type `ET_DYN` for the sBPF machine with `e_flags` equal to zero, exactly one `.text` section whose length is a multiple of eight, and `REL` relocations of the four kinds the sBPF linker emits. Anything else is refused with the name of the unsupported feature: `RELA` sections, relocations at unaligned text offsets, segments with uninitialized memory, a non-empty `.eh_frame`.

Before any rewrite, the input is loaded by the Agave runtime's own loader with its syscall registry, and its text is checked by the virtual machine's verifier [5]. After the last rewrite the text is verified again, and the rebuilt ELF is loaded again. A file that the validator would not deploy cannot leave the stage.

An instruction occupies one eight-byte slot, except the 64-bit immediate load, which occupies two. The stage numbers slots from zero and calls a slot index a program counter, *pc*.

7.2 Roots, functions and protected slots

A first pass over the section, symbol and relocation tables produces four sets.

- **Roots**: slots at which execution may begin without a direct branch or call leading there. These are the ELF entry point, every function symbol in the dynamic symbol table, the target of every relocation that stores a code address (in text or in data), and the target of every 64-bit immediate load whose value is an address inside `.text`.
- **Functions**: every function symbol in either symbol table, every relocated call target, and every code address a relocation or an immediate load refers to.

- **Calls**: for each call instruction with a relocation, its resolved target.
- **Protected slots**: every slot a relocation patches, and the second slot of a relocated 64-bit load. The loader will overwrite part of these slots at load time, so no rule may rewrite them.

7.3 Rewrite rules

The rules operate on single instructions or adjacent pairs. None applies to a protected slot. We write r_d and r_s for the destination and source registers of an instruction and *imm* for its immediate. Rules 1 and 2 are applied only when the program contains no register-indirect call, since their side conditions depend on knowing every way control can reach a slot.

Rule 1 (Unreachable code)**.** Let L be the set of slots reachable from the roots by following fall-through, both arms of conditional branches, the targets of unconditional branches, and the targets of direct calls together with their return points. An instruction at a slot outside L is replaced by `exit`; a two-slot load is replaced as a pair.

Rule 2 (Stack reload forwarding)**.** The pair

$$\texttt{stxdw [r10+}o\texttt{], }r_s\texttt{ ;}\quad \texttt{ldxdw }r_d\texttt{, [r10+}o\texttt{]}$$

with equal offsets o becomes the same store followed by `mov64` r_d`,` r_s, provided no branch, call or root targets the second instruction. The load then cannot be entered with a different value in the stack slot, and a full-width store followed by a full-width load of the same slot yields the stored register. A load that would fault is preceded by a store that faults identically.

Rule 3 (Strength reduction)**.** For an immediate $imm = 2^k$ with $0 \le k \le 30$:

$$\texttt{mul64 }r_d, 2^k \;\to\; \texttt{lsh64 }r_d, k$$

$$\texttt{div64 }r_d, 2^k \;\to\; \texttt{rsh64 }r_d, k \qquad \texttt{div32 }r_d, 2^k \;\to\; \texttt{rsh32 }r_d, k$$

$$\texttt{mod64 }r_d, 2^k \;\to\; \texttt{and64 }r_d, 2^k - 1 \qquad \texttt{mod32 }r_d, 2^k \;\to\; \texttt{and32 }r_d, 2^k - 1$$

Division and remainder in sBPF version 0 are unsigned, and the 32-bit forms zero-extend their result in both the original and the replacement. A 32-bit multiply is left alone: in this version it sign-extends its result, which a shift would not reproduce.

Rule 4 (Constant branches)**.** A conditional branch that compares a register with itself is decided statically: `jeq`, `jge`, `jle`, `jsge` and `jsle` are always taken; `jne`, `jgt`, `jlt`, `jsgt` and `jslt` never are. `jset` against the immediate zero is never taken. A taken branch becomes an unconditional jump to the same target and an untaken one a jump of displacement zero.

Rule 5 (Jump threading)**.** Let $\mathit{thread}(p)$ follow unconditional jumps from slot p until it reaches an instruction that is not one, returning p unchanged if the chain leaves the text or returns to a slot already visited. A branch at slot q with target p is retargeted to $\mathit{thread}(p)$ when that differs from p, is not q itself, and the new displacement fits the 16-bit field. A direct call is retargeted in the same way, except that a call whose immediate is -1 is never touched and never produced: that value is the loader's marker for an unresolved symbol.

Rule 6 (Identity runs)**.** An instruction is an identity when it leaves every register and all memory unchanged: `mov64` r, r; `add64`, `sub64`, `or64`, `xor64`, `lsh64`, `rsh64` or `arsh64` with immediate 0; `mul64` or `div64` with immediate 1; `and64` with immediate -1. In a maximal run of $m \ge 2$ consecutive unprotected identities that is followed by another instruction, the first is replaced by a jump over the remaining $m - 1$. The 32-bit forms are never identities, because they clear the upper half of the destination.

The rules are applied in the order given, each as one pass over the text. Rule 6 leaves the skipped identities in place as unreachable slots, which compaction later removes when it may.

7.4 Bounding indirect calls

Removing code from the file requires knowing that a slot cannot be reached. Direct control flow is explicit, but sBPF also has a register-indirect call, `callx`, which in version 0 accepts any in-range instruction address as its target. Rust uses it for trait objects and function pointers, and programs built with the standard library contain it. A sound rewriter must either keep every slot that a `callx` might reach or prove where each `callx` can go.

The stage runs an abstract interpretation over the whole program to attempt the proof. Its abstract value is either *unknown* or a set of at most 16 atoms. An atom is a number tagged with a provenance bit, or a stack address given as a frame and an offset. The provenance bit is set exactly when the number was produced by a relocated load, or read from an eight-byte cell of read-only data that a relocation fills. Such numbers are addresses the linker put there. A number with the bit clear is an ordinary constant, whatever its value.

The abstract state holds a value for each of the eleven registers and for up to 128 stack cells. The transfer functions are deliberately narrow:

- A relocated 64-bit load yields a pointer; an unrelocated one a plain number.
- Register moves copy values. Additions and subtractions of a zero keep the provenance bit; every other arithmetic result is a plain number, and operations outside a small set yield unknown.
- A full-width store to a single known stack cell records the value and invalidates overlapping cells. A store through any other address clears the remembered stack. A full-width load from a known stack cell or from `.rodata` or `.data.rel.ro` yields the stored or relocated value.
- Syscalls are modeled by name. Logging and return-data syscalls leave memory alone, the memory-copy and memory-set syscalls transfer or set whole cells when their arguments are known, the aborting syscalls end the path, and any other syscall clears the remembered stack.
- Direct calls are followed into the callee with a fresh frame, one context per callee, and return values flow back to every recorded call site with the callee-saved registers restored.

At a `callx` the analysis inspects the value of the target register. The site is *resolved* when the value is a set in which every atom that denotes a text address carries the provenance bit; its targets are those addresses. If the value is unknown, or contains a stack address, or contains a text address without provenance, the site is unresolved: an address computed by arithmetic is never accepted as proof of a call target. The analysis is also bounded in work and in contexts, and it gives up on recursion through a frame already on the abstract call stack. Giving up marks the whole analysis as limited.

The result decides how much the stage may remove:

$$\text{live slots} = \begin{cases} \text{the slots the analysis reached} & \text{if every site is resolved,} \\ \text{every slot} & \text{otherwise,} \end{cases}$$

where a limited analysis counts as an unresolved site.

The report lists each indirect call site with its program counter, whether it was resolved, its target set or the reason it was not, and the number of slots retained on its account.

7.5 Compaction

Compaction rebuilds the ELF file around the surviving slots. With $keep(p)$ true for live slots, the new position of a kept slot is the number of kept slots before it, giving a partial map μ from old to new program counters. The stage then rewrites every reference through μ:

- branch displacements and unrelocated call immediates are recomputed from the mapped positions of their source and target, and compaction fails if a displacement no longer fits or a call would become the -1 marker;
- the ELF entry point, every symbol's value and every text symbol's size are mapped, and symbols whose code was removed are cleared;
- relocation entries are moved to the new offsets of their sites, and entries whose site was removed are dropped;
- the dynamic section's addresses and table sizes are recomputed, and section and program headers are laid out again with each allocated section's address equal to its file offset, as version 0 requires.

Only relocation records identify which immediates are addresses. A numeric constant that happens to fall in the program's address range, such as a comparison threshold in wide division, is left unchanged.

Function merging. When every indirect site is resolved, functions with identical bodies are merged. A function body is the kept slots from one function entry to the next. Its key is the sequence of its instructions with intra-function branch displacements renumbered over kept slots, relocated immediates replaced by a canonical description of what the relocation refers to (an address, or a symbol name for a syscall), and direct call targets replaced by the callee's identity after earlier merges. A body is eligible when all its branches stay inside it and it ends in `exit`. Roots are never merged away, which preserves the identity of any function whose address the program can observe. Bodies with equal keys and equal length share one copy; μ sends each slot of a merged body to the corresponding slot of the survivor.

Prelinking. An sBPF version 0 program is loaded at a fixed virtual base address. Relocations that store an absolute address, whether in a 64-bit load or in a data cell, can therefore be applied ahead of time: the stage writes the final address into the instruction or the cell and removes the relocation entry. Relocations that bind a call to a syscall or to a function by symbol are kept, since the loader resolves those through its registry. Each removed entry shortens the relocation table by 16 bytes.

Sections that are not loaded. Debug and comment sections that are not part of the loaded image are emptied.

The report records the hashes and sizes of the input and output, the count of each rewrite, the roots, the list of modified program counters and μ as a list of affine runs $(old, new, length)$, so that a trace taken on one file can be read against the other.

7.6 How the rules are tested

Each rule ships with tests that execute code before and after the rewrite on the production interpreter of the validator's virtual machine and compare complete machine states. A test builds a small instruction sequence that triggers a rule, applies the stage, and runs both versions from 54 initial states: six chosen edge values (0, 1, $2^{64}-1$, 2^{63}, $2^{32}-1$ and `0xffffffff00000000`) and 48 further seeds, each of which fills ten registers by a mixing

function and fills the stack, heap and input regions with seed-dependent bytes. After execution it compares all register values, the full contents of the input region, the stack and the heap, and the result of the run, including faults.

Property 1 (Checked for each rule)**.** For every initial state tested, the rewritten code ends in the same registers, the same memory and the same result as the original, and executes no more instructions. For a rule intended to save work, it executes strictly fewer on at least one state.

Negative tests fix the side conditions: a stack reload that is a branch target is not forwarded; a 32-bit operation with an identity-looking immediate is not skipped; a program with an indirect call keeps every possible target; an address derived by arithmetic stays unresolved; a partial overwrite of a stack cell invalidates a function pointer held there; two functions that differ only in the literal a relocation refers to are not merged. Structural tests build complete ELF files with symbols, data relocations and function pointers, compact them, load the result with the Agave loader and execute both files. Beyond the rule tests, whole programs are run through the differential harness of Section 8 as controls, comparing a build with its own compacted form.

8 The Differential Proof

`tare-prove` decides the equivalence of Definition 5 by execution. It runs complete, signed transactions against the two builds and compares what each leaves behind. Differential testing of this kind, in which two implementations of one specification check each other, goes back to McKeeman [24]; here the two implementations are two compilations of one source, and the oracle for each is the other.

8.1 The execution environment

Both builds run in LiteSVM [17], an in-process Solana virtual machine built on the Agave runtime [3]: the same loader, verifier, interpreter, syscalls, metering and post-execution account checks a validator applies, without consensus or networking. For each sequence of the corpus the harness constructs one machine with the sequence's starting accounts and then clones it. One copy loads the stock build at the program's address and the other loads the Tare build at the same address. Everything else is shared by construction: built-in programs, the SPL programs, sysvars, the fee payer's balance and the metering configuration.

Programs the target calls are loaded identically on both sides. Sibling programs of the same workspace are compiled once with stock Anchor and the same ELF is given to both machines; SPL programs are those the SVM ships. A difference between the two runs can therefore originate only in the program under test.

8.2 The three corpora

A corpus has three parts, counted separately in every report.

Tests. The program's own test suite is the best statement of what its authors expect it to do. Tare records it in the form of transactions. The suite is run unchanged against the stock build on a local validator, behind a recording proxy that captures every transaction the tests submit: the signed transaction bytes, whether it landed or was rejected in preflight, the state of every account it names before and after, the slots, and the validator's outcome (error, logs, return data and compute units). The executable programs the tests deploy are captured with their hashes.

Replay treats a recording as a sequence of frames. For each frame the harness builds a fresh machine for each build from that build's current state, executes the recorded transaction (as a simulation if it was rejected in preflight, so that no fee is committed), and compares observations. Between frames, anything in the recorded pre-state that the previous frame's post-state does not explain is an input from the environment, such as an airdrop, a sysvar that advanced or state set up by the test script; it is applied identically to both builds. Changes made by the program are never overwritten: each build's own results carry into its next frame.

The replay also reports a second, separate measurement: whether the stock build under LiteSVM reproduces what the validator recorded for each frame. Equivalence of the two builds does not depend on it, and it is reported as its own count with the first difference.

Generated. Sequences constructed from the program's IDL, described in Section 8.3. They are deterministic: the same IDL, source and seed produce the same corpus.

Mainnet. `tare prove --program <address>` samples the program's most recent transactions from an RPC endpoint, three by default and at most twenty. For each signature it fetches the transaction and keeps it if the program is invoked at the top level or in an inner instruction. It then takes a snapshot of every account the transaction loads, including accounts reached through address lookup tables and the Clock and SlotHashes sysvars, and fetches the ELF of every other program the transaction touches. Legacy and versioned messages are accepted. Transactions that failed on chain are kept and compared like any other.

Each sampled transaction is replayed on both builds over the account state fetched when it was sampled. The comparison shows that on mainnet instruction data, account contents and cross-program calls the two builds leave the same observation.

8.3 The generated corpus

The generator's input is the stock IDL, which names every instruction with its arguments and accounts, and a set of *source hints* extracted by parsing the program's unchanged source: which accounts an instruction initializes, which are optional or required to be zeroed, the payload type of each typed account, composite account groups, token and mint relationships expressed in constraints and CPI structures, program-derived address seeds the IDL omits, and the types of accounts read from the remaining-accounts list.

Identities and values. The generator is seeded with a 64-bit constant and runs four rounds. In round j every named account and every address-typed argument receives a keypair derived as

$$\mathit{seed} = \text{SHA-256}(\texttt{"tare:"} \,\|\, s \,\|\, \texttt{":"} \,\|\, j \,\|\, \texttt{":"} \,\|\, \mathit{name}),$$

so every address is reproducible and every signer can sign. Integer arguments take the round's value from $(0, 1, 42, 255)$, booleans alternate, strings are `"tare-`j`"`, options are absent and vectors empty; composite types are encoded field by field from the IDL's type definitions, with generic parameters substituted. Program-derived addresses are computed from their IDL or source-derived seeds, and a bump argument receives the canonical bump of the address it belongs to. Signers are funded. Token mints, token accounts and associated token accounts are created by real setup transactions to the System, Token and Associated Token programs placed in the sequence before the calls that need them; accounts a client could not create, such as program-derived vaults, start from a typed snapshot, and the sequence records which accounts have which origin.

Case	Construction
`short-instruction`	instruction data cut to seven bytes
`unknown-discriminator`	eight bytes of `0xff`
`short-arguments`	the discriminator with no arguments
`alternate-<arg>`	a bump or nonce argument set to the round value where the canonical bump was used
`missing-accounts`	the account list emptied
`missing-signer-`i	signer flag cleared on account i
`multiple-faults-`i	missing signer and a typed account truncated to three bytes
`readonly-`i	writable flag cleared on account i
`invalid-account`	for each typed, non-initialized account: owner replaced; data replaced by eight `0xff` bytes; data truncated to three bytes
`wrong-seeds-`i	a program-derived account replaced by a fresh address carrying a copy of its contents
`optional-none-`i	an optional account replaced by the program's own address
`remaining-truncated-`k	the remaining-accounts list cut to k entries
`duplicate-account`	the first account appended again
`duplicate-declared-`i-j	two writable accounts given the same address
`wrong-declared-program-id`	the same call sent to the program loaded at a second address
`seed-`j, `repeat-1`, `repeat-2`	the well-formed call, committed three times

Table 3: Cases generated for each instruction in each round. All but the last row are speculative.

Sequences. A sequence is the starting accounts, the signer seeds and an ordered list of cases. Instructions are ordered so that those that zero or initialize accounts come first, and each round emits the whole program in that order. For small programs with several non-initializing instructions, further sequences run each of them first after initialization, so that an instruction that closes or burns an account does not hide the others. Independent sequences then exercise each instruction in isolation over its own typed fixtures, and again with argument values, payers and owners resolved from the source, so that unrelated instructions that reuse a field name do not share state.

Cases. For every instruction the generator emits the well-formed call and a family of malformed ones. Table 3 lists them. A case is either *committed*, in which case its effects carry into the rest of the sequence, or *speculative*: it runs on a copy of both machines' state, is compared, and is discarded. All malformed cases are speculative and are placed before the well-formed call, so that an initializing instruction still meets an uninitialized account. The well-formed call is committed and then repeated twice, which exercises the paths taken when an account already exists.

When a transaction is wrong in two ways, the order of the framework's checks decides which error is reported; `multiple-faults` checks that both builds make the same choice. The duplicate cases exercise the aliasing rules of Sections 6.2 and 6.5. The wrong-program-id case loads both builds at a second address derived from the program's and checks the declared-id guard.

Built-in instructions. Each sequence also addresses the framework's own entry points. For programs with the on-chain IDL instructions, the generator runs their whole lifecycle as committed cases (create, resize, create buffer, write, set authority, write under the new authority, set buffer, close) with speculative variants at each step: a missing last account, missing signers, read-only accounts, wrong owners, short accounts, bad discriminators, a wrong base address, an aliased buffer, invalid buffer lengths and closing an account into

itself. Fourteen malformed IDL payloads and a call to the event-CPI entry with no accounts complete the set.

8.4 Execution and comparison

For each case the harness clones both machines, applies the case's account mutations to both, builds one message with the sequence's fee payer, signs it with the keypairs whose signatures the message requires, and submits the identical signed transaction to each machine. It then compares the two observations of Definition 4 over the tracked accounts and increments a counter for each kind of comparison made: one per input for logs, return data and errors, and one per tracked account per input for bytes and for lamports. Ownership, the executable flag and the rent epoch are compared in the same pass over the accounts.

Errors are compared as the runtime's full transaction error, which includes the index of the failing instruction and the custom error code. Logs are compared after removing lines that match the runtime's compute report exactly, a line of eight space-separated tokens of the form `Program`, a valid address, `consumed`, an integer, `of`, an integer, `compute`, `units`. A program log that merely resembles this is kept.

The first input on which any component differs becomes the report's *first divergence*: the list of differences with both values, the signed transaction, and the complete pre-state of every tracked account. That is sufficient to replay the input in isolation, and it is the format `tare flag` accepts. The harness continues through the rest of the corpus so that counts and outcomes are complete, and the command then exits with a failure and prints the divergence.

8.5 Measurement

Weights fall out of the same run. Every successful single-instruction case contributes one sample to its instruction on each build, and the weights are the upper medians of Definition 2. For recorded transactions, the units of the target instruction are parsed from the runtime's `invoke`, `consumed` and `success` lines with a stack that mirrors the call depth; a recorded transaction contributes a sample only when it consists of one instruction addressed to the target and that invocation has exactly one consumed line, whose value includes the cost of the calls it makes. Setup transactions are compared and are not samples. Every transaction is a proof input whether or not it is a sample.

8.6 The replay bundle and the corpus hash

The report contains the whole corpus. Its *replay bundle* holds

- the path of the program's manifest relative to the repository root, and the Cargo features selected;
- the generated corpus in full: seed, sequences, starting accounts, signer seeds and every case;
- the recorded test corpus and the sampled mainnet corpus, when present, with their transactions and account snapshots;
- every external program used, as ELF bytes with its SHA-256;
- the supplied IDL, when one was used.

The bundle is serialized as JSON, compressed with gzip and embedded in the report in base64 together with the SHA-256 of the uncompressed bytes.

Definition 6 (Corpus hash)**.** The corpus hash of a proof is the SHA-256 of its serialized replay bundle.

A record stores this hash. Since the bundle includes the generated transactions themselves, checking a record does not depend on the generator producing the same corpus again, and since it includes account snapshots and external programs, checking does not depend on any network state. `tare check` unpacks the bundle, verifies its hash and every external program's hash, rejects manifest paths that are absolute or leave the repository, and replays exactly these inputs.

The public report is the proof document without two diagnostic sections (the bytecode stage's report and the recorded-corpus provenance). It carries the program's name and address, the toolchain and Anchor versions, the build environment, the account representation, the source repository and commit, both builds with hash, size and rent, the weights of every instruction, the totals, the proof status with corpus hash, counts and comparison counters, and the replay bundle.

9 The Registry

The registry is a program on Solana whose accounts say, for a program address, which two builds were compared, from which source, over which corpus, with what result, and who said so. It is an ordinary Anchor 1.2.2 program of four instructions, built with `tare build` and deployed from its Tare build.

9.1 Accounts

All three account types are program-derived, so an address can be computed from what it describes. Field-level layouts are in Appendix A.

Config. One account at the address derived from the seed `"config"`. It holds the admin, the fee recipient and the attest fee in lamports, in 80 bytes.

Record. One account per attestation, derived from

$$(\texttt{"record"},\ \mathit{program},\ \mathit{report_hash},\ \mathit{attester}).$$

It holds the address of the program the builds are of, the SHA-256 of the stock build and of the Tare build, the source repository and commit, the toolchain version, the corpus hash and the three corpus counts, the gross and net totals, the report hash and location, the attester and the slot. A record is allocated at its maximum size of 1,128 bytes.

Because the report hash and the attester are seeds, two different reports about one program, or the same report attested by two parties, are distinct records. Attesting the same report twice from the same key fails at account creation, before any fee moves.

Flag. One account per filed input, derived from

$$(\texttt{"flag"},\ \mathit{record},\ \mathit{input_hash}).$$

It holds the record's address, the SHA-256 of the input document and its location, the flagger and the slot, and is allocated at its maximum size of 628 bytes. One input can be filed against a record once.

9.2 Instructions

`init_config(admin, fee_recipient, attest_fee_lamports)`
Creates `Config`. The signer must be the registry program's current upgrade authority: the instruction takes the program account and its ProgramData account, checks that the first refers to the second, and checks that the ProgramData's upgrade authority is the signer. The admin it installs may be a different key and does not sign.

`set_config(new_admin, fee_recipient, attest_fee_lamports)`
Replaces all three fields. The signer must be the current admin.

`attest(args)`
Creates a `Record` and pays the fee. Described below.

`flag(input_hash, input_location)`
Creates a `Flag` for an existing `Record`. The record account must be owned by the registry and deserialize as a record, so a flag cannot point at anything else.

Every instruction emits an event through Anchor's event CPI: the program invokes itself with its event-authority address as signer and the event as instruction data, which places the event in the transaction's inner instructions where it is retained in full. `RecordAttested` carries every field of the record, the fee recipient and the fee paid; `RecordFlagged` every field of the flag; `ConfigInitialized` and `ConfigUpdated` the three configuration fields. In events the source commit is rendered as lowercase hexadecimal text.

The program validates what it can validate on chain: the repository string is 1 to 256 bytes, the commit is 20 or 32 bytes (a SHA-1 or SHA-256 object id), the toolchain version is 1 to 64 bytes and locations are 1 to 512 bytes. A record is an attester's signed statement, and `tare check` tests it.

9.3 Attesting

`tare attest` is the only writer of records the toolchain provides, and it refuses to write unless the local state supports the statement. In order, it requires that

1. the build was not made with the rehearsal defect switch used to test `tare flag`;
2. the proof's status is `matched`;
3. the git working tree has no uncommitted or untracked changes;
4. the proof's repository and commit equal the checkout's origin and `HEAD`, and the repository URL carries no credentials or query;
5. the stock and Tare artifacts on disk hash to the values in the proof.

It then reads `Config` from the chain, checks its discriminator, and prints the fee it is about to pay. It uploads the report to the API, which stores it under its content hash (Section 9.8) and returns that hash with a location; the CLI recomputes the hash locally and stops if they differ. Finally it derives the record's address, sends `attest` signed by the team's keypair, waits for confirmation, and prints the record's address, the transaction signature and the record's page.

Figure 3 shows this path and the two that read a record back.

9.4 Checking

`tare check <record>` takes a record's address and nothing else. It reads the account, verifies the owner and discriminator, and decodes the fields. It fetches the report by its hash and verifies the hash of what it received. It then requires the report and the record to

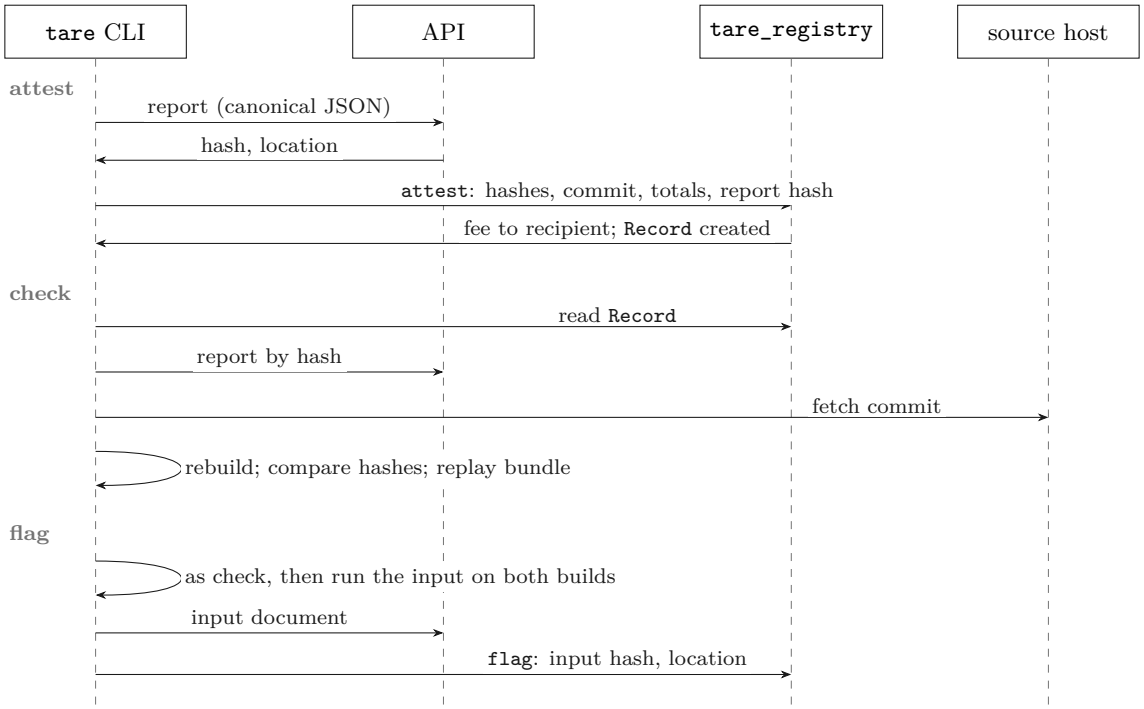


Figure 3: The three registry flows. Reports and flag inputs are stored off chain by content hash; the chain holds the hashes.

agree on the program address, both build hashes, the corpus hash, the source commit and repository, the toolchain version, the three corpus counts and both totals.

With a consistent record in hand, it fetches the recorded commit into a fresh directory with its submodules, confirms that the checkout is at that commit and clean, and compares the recorded build environment with its own. It rebuilds with the recorded features and, when the record used one, the supplied IDL from the bundle, and requires the rebuilt stock hash and Tare hash to equal the record's. It then replays the bundle on the rebuilt artifacts and requires a matched proof whose build hashes and corpus hash equal the record's. Its output ends with the record, the commit, both hashes, the number of transactions rerun and the word `matches`, or with the first field that differed.

A successful check establishes three things independently of the attester: the recorded source at the recorded commit compiles to the recorded stock build and the recorded Tare build; those two builds are equivalent over the recorded corpus; and the weights in the record are what that corpus measures.

9.5 Flagging

`tare flag <record> --input <file>` files an input on which the two recorded builds differ. The input file holds a signed transaction and the accounts it starts from, in the format of a report's first divergence. The command performs the rebuild of `tare check`, so that it holds the two recorded builds, and executes the input on both. If the observations are equal it stops and says so; nothing is uploaded and no transaction is sent. If they differ it uploads the divergence document, derives the flag's address from the record and the document's hash, and sends `flag`.

A flag is therefore a reproducible counterexample attached permanently to the record it concerns. The record's page lists its flags with their inputs, and anyone can rerun one.

9.6 The fee

Building, weighing, proving and checking are free. Attesting pays a fee in SOL, the asset a team already holds to deploy its program. The amount and the recipient are fields of `Config`. `attest` requires the fee recipient account it is given to be the one `Config` names, and transfers `Config.attest_fee_lamports` from the attester to it through the System Program in the same instruction that creates the record. The record's event states the recipient and the amount, so the fees received are the sum of one field over the registry's events.

The fee recipient is the launch wallet of the product's token, TARE, which has a fixed supply of 1,000,000,000 and is distributed by fair launch. A crank on the product's server signs as that wallet. On each interval, when the wallet's balance exceeds a reserve plus one *clip*, a fixed amount of SOL per buy, it requests a quote for one clip of SOL into TARE from Jupiter's swap API [12] and submits a single transaction containing the swap and a burn of the quote's minimum output. On later ticks it reads each confirmed swap's own token balance change and burns whatever that swap delivered beyond the minimum. Every submission is journaled under an idempotency key (the mint and the interval window for a buy, the swap's signature for its second burn), so a restart cannot repeat a buy or burn the same tokens twice.

9.7 State on mainnet

The registry was deployed to Solana mainnet on 22 November 2026 (UTC). Its addresses are:

Program	`7viqDjx5k8ffnfnZiBv9dMhxzVhcGSYLMb5bKdDP1XtP`
`Config`	`Fvcic6U24RfK7CG9a4rKNaRubMgsQZN3NpywXEoYUQkD`
Fee recipient	`EvycjhbgcQDVN92xpPsdy6oLKezCcDiykg96g8ZXZDe9`

The deployed ELF is the registry's own Tare build, made with toolchain 0.7.0 in the container image: 271,672 bytes, against 278,048 for its stock build, with SHA-256 `348a6403b4fdadc36208769edf040c2e4aafc4f8c7967371ca48fa639b919cf5`. Read at slot 472,653,569 on 22 November 2026 (UTC), `Config` named the fee recipient above and an attest fee of 50,000,000 lamports. The CLI and the documentation read the current values from `Config`.

9.8 Indexing and the API

An indexer follows the registry program's transactions at finalized commitment, decodes the four events with the program's IDL and writes records, flags and fees to a database. The API serves them: the list of records, newest first and filterable by program; one record with its report and flags; registry totals; reports and flag inputs by hash. Uploads are limited to one mebibyte and stored under the SHA-256 of their canonical form, in which object keys are sorted recursively, so the same document always has the same address. The site's registry pages are views over these endpoints.

10 Weighing Deployed Programs

The weights of Section 3 need source. A deployed program can be weighed without it, from the chain alone, in a different sense: what its instructions cost on mainnet today. The site's `/weigh/<address>` page shows this for any program, and the same data is served by the API. It gives the gross side of the picture for a program whose team has not yet built with Tare, and it supplies the call frequencies the per-instruction weights leave out.

10.1 Sampling

A weighing is a bounded read. For a program address the service

1. reads the 36-byte program account and, without transferring its contents, the size of the ProgramData account it points to; the binary size is that size less the 45-byte header, and the rent shown is the rent-exempt minimum of the program account and the ProgramData account together;
2. looks for the program's IDL on chain, first at Anchor's IDL account for the program and then in the program-metadata form, to name instructions;
3. pages backward through the program's recent signatures, 100 per page for at most ten pages, skipping signatures whose transaction failed, and fetches transactions in batches of 20 until it has the configured number of measurable ones (50 by default) or has fetched 300.

10.2 Attribution

Compute is attributed from the transaction's log, which the runtime structures as nested `invoke`, `consumed` and `success` or `failed` lines. The service walks the log with a stack, matching each `invoke` line to the next instruction of the transaction's instruction tree at the same depth and under the same parent. Each successful invocation of the target program, at any depth, yields one measurement: the first eight bytes of its instruction data and the units on its `consumed` line, which include the calls it made. A transaction whose log does not match its instruction tree exactly is discarded whole. A log the runtime truncated keeps the invocations that completed before the cut.

Measurements are grouped by discriminator. For each group the service reports the number of calls, the 50th and 90th percentiles by nearest rank,

$$q_p = u^{(\lceil p\,k \rceil)} \quad \text{for } p \in \{0.5,\, 0.9\},$$

over the k sorted values, the total units, and the group's share of all units measured for the program. Groups are named from the IDL when one exists and shown by discriminator otherwise.

A program is marked as an Anchor program when an IDL is found, or when an invocation logs `Instruction: <Name>` and the SHA-256 of `global:<name>` begins with that invocation's discriminator. The second test recognizes Anchor programs that publish no IDL without relying on the log text alone.

10.3 States and caching

The chain is read only on request: a request for a program that has no stored weighing starts one in the background and answers `sampling` with a retry interval. The stored result is one of `ready`, `not_program` for an address that is not executable, `no_transactions`, or `calls_failed` for a program whose recent calls all failed, which is distinguished by checking that failed transactions did invoke it. A stored weighing is served until it is older than the cache age (900 seconds by default, eight times that for a program with nothing to measure).

When the registry holds records for the address, the response lists them, which connects the two kinds of weighing: what the program costs as deployed, and what a recorded build of its source costs.

GitHub repository	Commit	Anchor	Programs
`solana-foundation/anchor`	`8d9d1488d9b8`	1.2.2	70
`solana-foundation/anchor`	`0e115d4bd34b`	0.32.3	58
`solana-foundation/anchor`	`47284f8f0b98`	0.31.1	57
`Squads-Protocol/v4`	`3a39735d0c56`	0.32.1	1
`marinade-finance/liquid-staking-program`	`6909edc329db`	0.31.1	1
`orca-so/whirlpools`	`cc683ec92acb`	0.32.1	1

Table 4: Sources of the measured corpus. The registry, the remaining program, is built from its own workspace on Anchor 1.2.2.

11 Evaluation

The evaluation asks what Tare does to programs that were not written for it. Every figure in this section is read from the benchmark's result file, produced by toolchain 0.7.0 on 2026-11-15.

11.1 Corpus and method

The benchmark manifest pins 238 programs by repository and commit; 189 of them build under both pipelines and form the measured corpus. Table 4 lists where they come from. 164 are the programs under the Anchor repository's `tests/` directory, written by the framework's maintainers to exercise its features one at a time and in combination; 21 are the tutorial programs under `examples/`. Both sets are taken at three release commits, one for each of the version lines 0.31.1, 0.32.3 and 1.2.2. The production set is Squads v4 (a multisig), Orca Whirlpools (a concentrated-liquidity exchange), Marinade (liquid staking) and the Tare registry itself.

Each program is built in place with `tare build`, weighed and proven. Execution uses LiteSVM 0.18.2 on the Agave 4.3 runtime. The 189 programs declare 1,323 instructions between them. The corpus reaches 1,187 of these with at least one successful call, from 30,722 successful samples in all, and those are the instructions that have weights.

11.2 Compute

Summed over the corpus, gross is 6,295,676 CU and net is 4,459,739 CU: 1,835,937 CU, or 29.2% of the stock builds' compute on these instructions, is tare. The sum weights each instruction by its cost, so the heaviest instructions dominate it. The per-instruction view is more representative of what a team will see on its own program: the median instruction loses 46.8% of its compute units, the middle half of instructions lose between 25.9% and 57.2%, and 525 instructions lose half or more. The smallest reduction is 0.9% and the largest 83.3%. Every one of the 1,187 measured instructions is cheaper in the Tare build.

Figure 4 plots each instruction's net weight against its gross weight. Gross weights span three orders of magnitude, from 185 to 120,985 CU. Instructions of a few hundred to a few thousand units, where the framework's fixed work is a large share of the whole, sit near or below the half line. The heaviest instructions approach the diagonal, since a hash, a signature check or a CPI into another program costs the same from either build. Figure 5 gives the distribution of reductions.

Table 5 breaks the result down by source and by version line. The three lines of the Anchor repository behave alike: medians of 48.9%, 48.4% and 48.7% on 0.31.1, 0.32.3 and 1.2.2. Figure 6 shows the agreement across the whole distribution; the three adapters share one core.

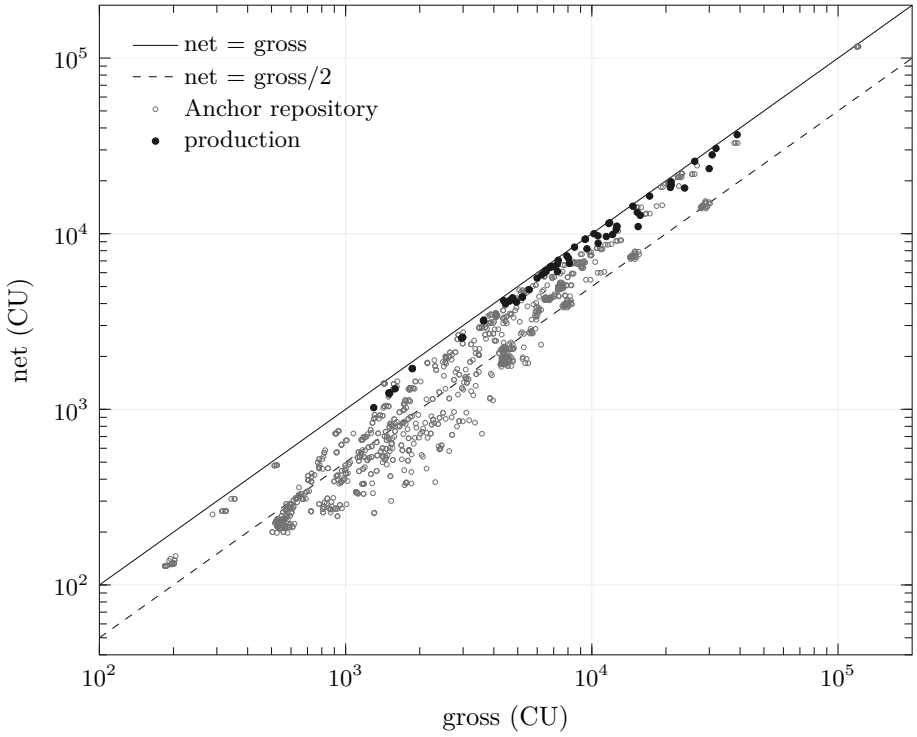


Figure 4: Net against gross compute for each of the 1,187 measured instructions, on logarithmic axes.

11.3 Binary size and rent

The 189 stock binaries total 38,786,032 bytes and the Tare binaries 30,970,216, a reduction of 7,815,816 bytes or 20.2%. The median program is 32.2% smaller. The two stages contribute separately. The linked output of the front stage totals 33,174,328 bytes, so replacing the framework's generated code removes 5,611,704 bytes; the bytecode stage then removes a further 2,204,112.

Rent follows size by Equation 1. Deployed side by side, the stock builds of the corpus would lock 197,199,143,320 lamports and the Tare builds 157,494,798,040, a difference of 39,704,345,280 lamports, or 39.70 SOL.

11.4 Production programs

Table 6 gives the four production programs. They differ from the repository's test programs in the direction Figure 4 predicts. Their instructions are heavy with their own logic (tick-array arithmetic in Whirlpools, stake accounting in Marinade, transaction and proposal handling in Squads) and with calls into the token and system programs, so the framework is a smaller share of each. The corpus reaches 34 instructions of Squads, 20 of Whirlpools and 6 of Marinade. Figure 7 shows Squads instruction by instruction.

The registry is the one production program on the 1.2.2 line and the one whose proof corpus is its own test suite: 17 recorded transactions covering authorization, admin transfer, exact fee and rent debits, a refused attestation to a wrong recipient, duplicate attestations and flags against missing records. Its four instructions weigh 84,919 CU gross and 65,361 CU net. The row describes the registry as benchmarked on 2026-11-15; the build deployed on 22 November is the one of Section 9.

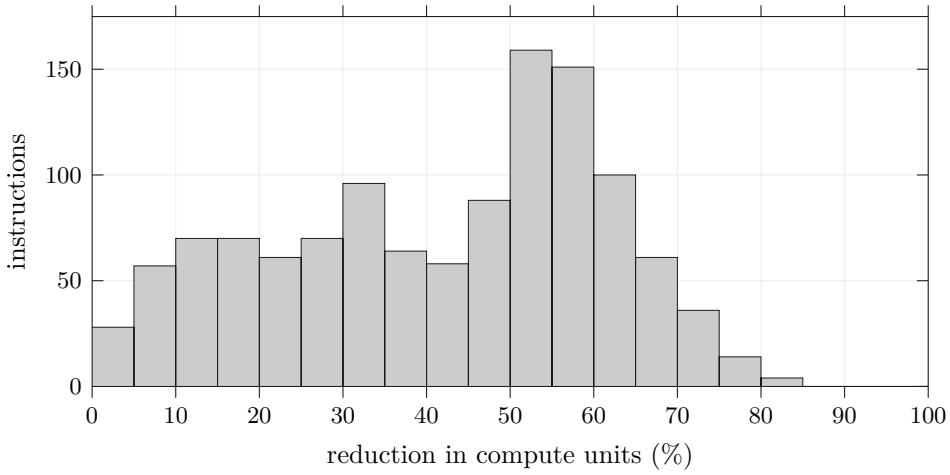


Figure 5: Distribution of the per-instruction reduction T_ι / G_ι in bins of five percentage points.

			compute units				binary bytes		
Cohort	prog.	instr.	gross	net	red. %	med. %	stock	Tare	transactions
Anchor `examples/`	21	45	136,117	72,562	46.7	58.5	3,342,752	2,158,880	16,476
Anchor `tests/`	164	1,078	5,509,653	3,804,228	31.0	48.4	31,848,088	25,527,552	240,630
Production	4	64	649,906	582,949	10.3	10.3	3,595,192	3,283,784	786
Anchor 0.31.1	58	362	1,893,002	1,317,985	30.4	48.9	13,674,744	10,744,408	93,557
Anchor 0.32.1	2	54	492,250	452,115	8.2	9.6	2,348,584	2,222,712	702
Anchor 0.32.3	58	369	1,819,112	1,271,305	30.1	48.4	12,961,712	10,213,016	95,370
Anchor 1.2.2	71	402	2,091,312	1,418,334	32.2	48.7	9,800,992	7,790,080	68,263
All	189	1,187	6,295,676	4,459,739	29.2	46.8	38,786,032	30,970,216	257,892

Table 5: Results by source and by Anchor version line. “red.” is the reduction of the cohort’s summed compute; “med.” is the median of the per-instruction reductions. Anchor 0.32.1 is the version Squads v4 and Whirlpools pin.

11.5 Distance to a hand-written program

The reference point for the net weight is what a careful developer would write by hand. The benchmark keeps hand-written Pinocchio [4] equivalents of the user instructions of three tutorial programs, written to follow the pinned Anchor source in validation order, account layout, custom errors, logs and state changes, and run through the same harness on the same generated inputs. Table 7 compares the three builds.

Instruction	stock	Tare	hand-written	gap closed (%)
`basic_0::initialize`	197	132	130	97
`basic_1::initialize`	4,460	1,852	1,674	94
`basic_1::update`	725	262	174	84
`basic_4::increment`	2,791	2,115	1,769	66
`basic_4::initialize`	9,236	6,373	6,232	95

Table 7: Compute units of the stock build, the Tare build and a hand-written Pinocchio program for the same instruction. The last column is (stock − Tare)/(stock − hand-written).

On these instructions the Tare build recovers between 66% and 97% of the distance from the stock build to the hand-written one, starting from source that was not changed.

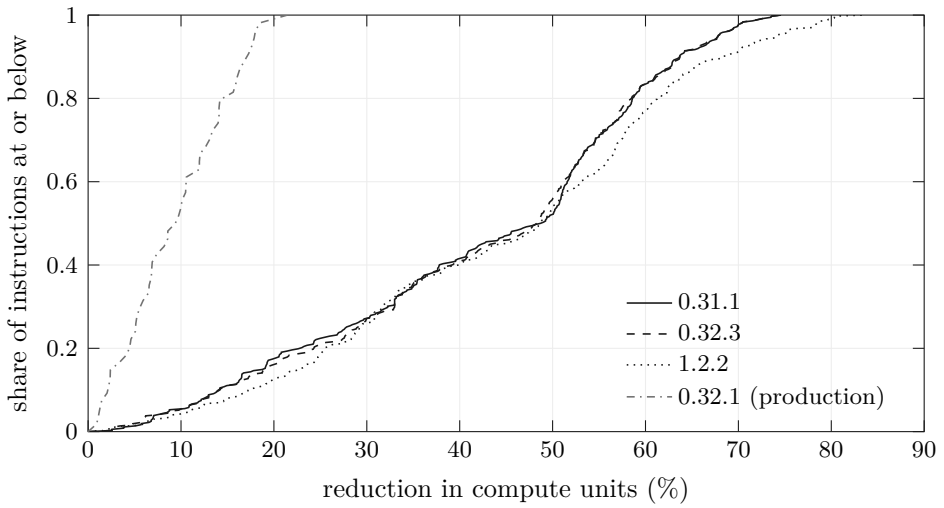


Figure 6: Empirical distribution of per-instruction reductions by Anchor version line.

Program	Anchor	instr.	gross	net	red. %	stock bytes	Tare bytes	tx
Squads v4	0.32.1	34	307,612	278,716	9.4	791,768	777,488	485
Orca Whirlpools	0.32.1	20	184,638	173,399	6.1	1,556,816	1,445,224	217
Marinade liquid staking	0.31.1	6	72,737	65,473	10.0	964,616	782,112	67
Tare registry	1.2.2	4	84,919	65,361	23.0	281,992	278,960	17

Table 6: The production programs. “instr.” counts instructions with weights; “tx” is the number of transactions compared.

11.6 The bytecode stage

The linked programs entering the stage contain 3,380,847 instructions and 231,621 relocation sites. The stage prelinked 137,757 address relocations across all 189 programs, which accounts for the 2,204,112 bytes it removes (6.6% of the linked size), and threaded 559 branch targets in 84 programs. The programs contain 8,533 register-indirect call instructions. The value-flow analysis examined 1,293 reachable call sites and resolved 313 of them to exact target sets, each listed in the program’s optimization report with its targets.

11.7 Proof volume and coverage

Across the corpus the harness executed 257,892 transactions on each build: 257,875 generated and 17 recorded. It made 9,713,524 comparisons, of which 4,340,978 were of an account’s bytes and as many of its lamports, and the rest of logs, return data and errors, one each per transaction. All matched: every one of the 189 proofs has status `matched` and no first divergence.

Successful calls are a minority of the corpus by design. Of the 257,892 transactions, 30,722 are successful single-instruction calls that serve as samples; most of the remainder are the malformed cases of Table 3, on which the two builds must fail identically.

Appendix B lists the 65 language constructs the measured programs use, with the number of programs exercising each on each version line, and Appendix C gives the result for every program.

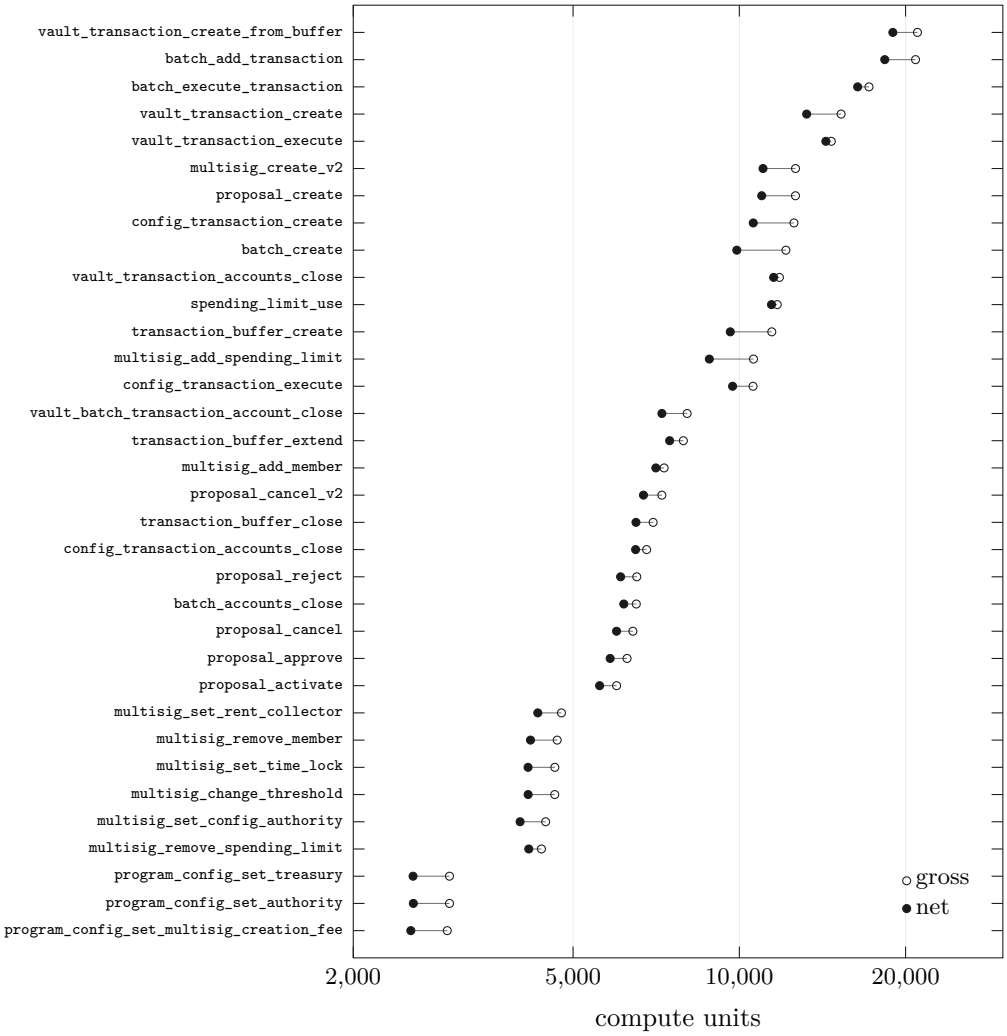


Figure 7: Squads v4 (commit `3a39735d0c56`): gross and net compute of each measured instruction, ordered by gross weight.

12 Related Work

Optimizing behind a fixed source language. Solar [10] lowers Solidity through intermediate representations that understand the language's own constructs. Tare takes its structure from Solar. The difference in setting is that an Anchor program's cost is decided less by code generation for Rust, which LLVM [14] already performs, than by the code the framework's macros expand to, so Tare's front stage reimplements the framework and its back stage works on the linked artifact.

Frameworks and libraries for Solana programs. Anchor [1] is the framework Tare takes as input, and Pinocchio [4] the library whose treatment of the loader's input its front stage follows. Quasar [7], Star Frame [34] and Typhoon [38] are frameworks designed for low compute from the start, each with its own programming model. These projects lower the cost of programs written for them. Tare addresses programs that are already written and is complementary to these frameworks: its bytecode stage operates on linked sBPF and does not depend on the framework that produced it.

Equivalence of a rewrite. The p-token rewrite of SPL Token was published with equivalence proofs by Runtime Verification [30], which establish the correspondence of two specific programs by formal means. Such proofs are constructed for one pair of programs by specialists. Tare's proof is of a different kind, a comparison by execution over a published corpus, and it is produced by one command for any Anchor program, with the corpus inside the report.

Differential testing. Running two implementations of one specification against each other and treating disagreement as a defect is due to McKeeman [24]. Csmith [42] applied it to C compilers with randomly generated programs, and equivalence modulo inputs [15] derives program variants that must agree on given inputs. Tare's use is closest to the original setting. The two artifacts are compilations of the same source by two implementations of one language; the inputs are transactions, generated from the program's interface description with a deliberate emphasis on malformed ones; and the compared output is the whole externally visible effect of a transaction. Measurement on identical inputs under a deterministic meter also avoids the environmental bias that makes timing comparisons fragile [25]: a compute unit count is a function of the program and its input.

Translation validation and verified compilation. Translation validation [28, 26] checks each run of a compiler by proving the output equivalent to the input, and has been carried to a verified operating-system kernel [32]. CompCert [16] proves the compiler itself once. Alive [19] and Alive2 [18] verify peephole optimizations and bounded translation steps of LLVM with an SMT solver. Tare validates each run, like translation validation, by execution where those systems use proof. Its bytecode rules are small, local and stated with explicit side conditions (Section 7.3), the form Alive showed to be amenable to mechanical proof, and Section 13 names that as a build.

Optimizing BPF bytecode. Peephole optimization dates to McKeeman [23], and superoptimization, the search for the shortest equivalent instruction sequence, to Massalin [21], later automated for peephole rules [6] and made stochastic [31]. For Linux eBPF, K2 [40] synthesizes smaller and faster packet-processing programs with formal equivalence and safety checks, Merlin [20] applies multi-tier optimization at the IR and bytecode levels, and EPSO [43] is a caching superoptimizer for BPF bytecode. These systems target kernel eBPF, whose programs must satisfy the in-kernel verifier [11] and run through kernel JIT compilers [27]. sBPF programs differ in what a rewriter must preserve. They are relocatable ELF files with dynamic symbols and syscall bindings, they may call through registers into any address of their own text, and their cost is a deterministic instruction count. `tare-opt` is built around those facts: relocation-aware compaction, an analysis that bounds indirect call targets, and admission by the validator's own loader.

Reproducible builds and attestations. That a compiler must be trusted along with the source is Thompson's observation [36], and diverse double-compiling [39] counters it by comparing the outputs of independent compilers. Reproducible builds [13] make a binary checkable against its source by anyone, and in-toto [37] records signed statements about each step of a software supply chain. A Tare record is a signed statement that binds a source commit to two reproducible binaries and to the corpus over which they were compared, kept on the same chain as the program it describes, and `tare check` is its verification procedure.

Analysis of Solana programs. VRust [9] detects classes of vulnerabilities in Solana programs by static analysis of their Rust intermediate representation, including missing

	Build	What it adds
1	The burn in the open	Every run of the crank shown in the product as SOL in, TARE burned and a transaction signature, beside the records.
2	The build on chain	Every weighing and record states which build the program at the address is running; `tare check --deployed` compares the bytes on chain with the record.
3	Tare in CI	`tare weigh --base` weighs a change against the branch it leaves; a GitHub Action comments the weighing on each pull request, fails on a regression and attests on a release tag.
4	The network weighing	The weighing taken of Solana itself: compute by program over sampled blocks, and the share of it that runs under a record.
5	Full-history replay	A fourth corpus: every transaction a program ever executed, replayed through both builds over the state it executed on.
6	Every framework, and no framework	The bytecode stage as a front door for native, Pinocchio and Anchor v2 programs, and for deployed programs lifted from the chain with no source.
7	Proven passes	Each rewrite rule carries a machine-checked proof against sBPF's semantics, with its hash in the report; new rules enter only when their proof checks.
8	Findings	`tare build` reports what its typed view of accounts and checks shows: an unconstrained owner, a signer never required, a seed that does not bind.

Table 8: The builds on the road, in order.

ownership and signer checks. Tare's front stage holds the same information in typed form for every instruction it compiles: which accounts are read and written and which checks guard them.

13 The Road

The project is built toward one goal:

> *Solana runs at the weight of its own logic.*

A team writes in the framework it trusts, builds with Tare and deploys the build that is in the registry. The goal has a threshold that can be measured from the chain: it is reached when more than half of the compute units Solana mainnet executes run in programs whose deployed build has a record in the registry.

Table 8 lists the builds on the way, in order.

Build 4 produces the figure the goal is stated in, so progress toward it can be read on any day. Builds 5 to 7 extend the proof to a program's whole history, to any linked sBPF program, and to machine-checked rules. Each rests on a component described in this paper: the recorded replay of Section 8.2, the framework-independent stage of Section 7 and the rule set of Section 7.3.

14 Conclusion

Tare removes the framework's overhead at build time, from unchanged Anchor source. The source is compiled against a second implementation of the language and rewritten at the

bytecode level, and the result is weighed against the stock build instruction by instruction. On 189 open-source programs the median instruction sheds 46.8% of its compute units and every measured instruction is cheaper, with 257,892 transactions compared across account bytes, lamports, account metadata, logs, return data and errors without a difference. The evidence for each build is a record on Solana mainnet that names its source, its two binaries and its corpus, and that anyone can verify with one command.

The toolchain installs with

```
curl -fsSL https://api.tare.cx/install.sh | sh
```

and the registry, the benchmark and the documentation are at `https://tare.cx`.

References

[1] Anchor. Anchor: Solana program framework. Documentation at `https://www.anchor-lang.com`.

[2] Anchor. Get started: Migrating from v1. Anchor Docs (v2). `https://v2.anchor-lang.com/docs/v2/get-started/migrating-from-v1/`.

[3] Anza. Agave: High-performance validator and RPC client for the Solana blockchain. `https://github.com/anza-xyz/agave`.

[4] Anza. Pinocchio: Create Solana programs with no external dependencies attached. `https://github.com/anza-xyz/pinocchio`.

[5] Anza. solana-sbpf: SBPF virtual machine. `https://github.com/anza-xyz/sbpf`.

[6] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 394–403, 2006. doi:10.1145/1168857.1168906.

[7] Blueshift. Quasar: A blazing fast Solana program framework. `https://github.com/blueshift-gg/quasar`.

[8] Borsh. Borsh: Binary object representation serializer for hashing. `https://borsh.io`.

[9] Siwei Cui, Gang Zhao, Yifei Gao, Tien Tavu, and Jeff Huang. VRust: Automated vulnerability detection for Solana smart contracts. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, pages 639–652, 2022. doi:10.1145/3548606.3560552.

[10] DaniPopes, djtodoro, and Georgios Konstantopoulos. Solar: 100× faster Solidity. Paradigm, 2026. `https://www.paradigm.xyz/writing/solar-100-faster-solidity`.

[11] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, pages 1069–1084, 2019. doi:10.1145/3314221.3314590.

[12] Jupiter. Solana swap API. `https://developers.jup.ag/docs/swap/v2`.

[13] Chris Lamb and Stefano Zacchiroli. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*, 39(2):62–70, 2022. doi:10.1109/MS.2021.3073045.

[14] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

[15] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 216–226, 2014. doi:10.1145/2594291.2594334.

[16] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[17] LiteSVM. LiteSVM: A fast and lightweight library for testing Solana programs. `https://github.com/LiteSVM/litesvm`.

[18] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, pages 65–79, 2021. doi:10.1145/3453483.3454030.

[19] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, pages 22–32, 2015. doi:10.1145/2737924.2737965.

[20] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. Merlin: Multi-tier optimization of eBPF code for performance and compactness. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*, pages 639–653, 2024. doi:10.1145/3620666.3651387.

[21] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, 1987. doi:10.1145/36206.36194.

[22] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference*, San Diego, CA, 1993.

[23] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965. doi:10.1145/364995.365000.

[24] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[25] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 265–276, 2009. doi:10.1145/1508244.1508275.

[26] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pages 83–94, 2000. doi:10.1145/349299.349314.

[27] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61, 2020.

[28] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*, Lecture Notes in Computer Science, pages 151–166. Springer, 1998. doi:10.1007/BFb0054170.

[29] Reproducible Builds. Definitions. `https://reproducible-builds.org/docs/definition/`.

[30] Runtime Verification. Solana SPL Token and P-Token program equivalence proof. `https://github.com/runtimeverification/solana-token/blob/proofs/RV_EQUIVALENCE_PROOFS_REPORT.md`.

[31] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 305–316, 2013. doi:10.1145/2451116.2451150.

[32] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, pages 471–482, 2013. doi:10.1145/2491956.2462183.

[33] Solana. Compute budget. Solana Documentation. https://solana.com/docs/core/fees/compute-budget.

[34] Star Atlas. Star frame: High performance, modular Solana program framework. https://github.com/staratlasmeta/star_frame.

[35] The Cargo Book. Overriding dependencies. https://doc.rust-lang.org/cargo/reference/overriding-dependencies.html.

[36] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984. doi:10.1145/358198.358210.

[37] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410, 2019.

[38] Typhoon. Typhoon: Solana Sealevel framework. https://github.com/aursen-labs/typhoon.

[39] David A. Wheeler. Countering trusting trust through diverse double-compiling. In *21st Annual Computer Security Applications Conference (ACSAC '05)*, pages 33–48. IEEE, 2005. doi:10.1109/CSAC.2005.17.

[40] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 50–64, 2021. doi:10.1145/3452296.3472929.

[41] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. Whitepaper. https://solana.com/solana-whitepaper.pdf.

[42] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 283–294, 2011. doi:10.1145/1993498.1993532.

[43] Qian Zhu, Yuxuan Liu, Ziyuan Zhu, Shangqing Liu, and Lei Bu. EPSO: A caching-based efficient superoptimizer for BPF bytecode. arXiv:2511.15589 [cs.SE], 2025. https://doi.org/10.48550/arXiv.2511.15589.

A Registry Reference

Program id `7viqDjx5k8ffnfnZiBv9dMhxzVhcGSYLMb5bKdDP1XtP`. Accounts are Anchor accounts: an eight-byte discriminator, then the fields in order, Borsh-encoded. Strings and byte vectors carry a four-byte little-endian length. Integers are little-endian.

Accounts

Account	Field	Type	Bound
`Config`	`admin`	address	
seeds: `"config"`	`fee_recipient`	address	
80 bytes	`attest_fee_lamports`	`u64`	
`Record`	`program`	address	
seeds: `"record"`,	`stock_build_hash`	32 bytes	SHA-256 of the stock ELF
`program`,	`tare_build_hash`	32 bytes	SHA-256 of the Tare ELF
`report_hash`,	`source_repository`	string	1 to 256 bytes
`attester`	`source_commit`	bytes	20 or 32 bytes
1,128 bytes	`toolchain_version`	string	1 to 64 bytes
	`corpus_hash`	32 bytes	SHA-256 of the replay bundle
	`tests_count`	`u64`	
	`generated_count`	`u64`	
	`mainnet_count`	`u64`	
	`gross`	`u64`	program total, CU
	`net`	`u64`	program total, CU
	`report_hash`	32 bytes	SHA-256 of the report
	`report_location`	string	1 to 512 bytes
	`attester`	address	the signer of `attest`
	`slot`	`u64`	slot of the attestation
`Flag`	`record`	address	an existing `Record`
seeds: `"flag"`,	`input_hash`	32 bytes	SHA-256 of the input document
`record`,	`input_location`	string	1 to 512 bytes
`input_hash`	`flagger`	address	the signer of `flag`
628 bytes	`slot`	`u64`	

Instructions

Each instruction also takes the event-authority address (seed `"__event_authority"`) and the program itself as its last two accounts, as Anchor's event CPI requires.

Instruction	Accounts, in order	Arguments
`init_config`	authority (signer, writable); registry program; its ProgramData; `Config` (created); System Program	`admin`, `fee_recipient`, `attest_fee_lamports`
`set_config`	admin (signer); `Config` (writable)	`new_admin`, `fee_recipient`, `attest_fee_lamports`
`attest`	attester (signer, writable); `Config`; fee recipient (writable); `Record` (created); System Program	the fourteen fields of `Record` before `attester`, in order
`flag`	flagger (signer, writable); `Record`; `Flag` (created); System Program	`input_hash`, `input_location`

Errors

Error	Raised when
`Unauthorized`	the signer is not the upgrade authority (`init_config`) or the admin (`set_config`)
`ProgramDataMismatch`	the ProgramData account is not the registry's
`RepositoryLength`	the source repository is empty or longer than 256 bytes
`CommitLength`	the source commit is not 20 or 32 bytes
`VersionLength`	the toolchain version is empty or longer than 64 bytes
`LocationLength`	a location is empty or longer than 512 bytes
`FeeRecipientMismatch`	the fee recipient passed to `attest` is not the one in `Config`

Events

`ConfigInitialized` and `ConfigUpdated` carry `admin`, `fee_recipient` and `attest_fee_lamports`. `RecordAttested` carries the sixteen fields of `Record`, with `source_commit` as lowercase hexadecimal text, followed by `fee_recipient` and `fee_lamports`. `RecordFlagged` carries the five fields of `Flag`.

B Constructs Exercised by the Corpus

Each row is a language construct found in the source of the measured programs. Each cell is the number of measured programs on that Anchor version line that use the construct, all of them with a matched proof.

Construct	0.31.1	0.32.1	0.32.3	1.2.2
`#[access_control]`	1	1	1	2
`Account`	33	2	33	39
`#[account]`	38	2	38	45
`account_constraints`	36		37	43
`AccountLoader`	6	1	6	5
`address`	4	2	3	5
`AnchorDeserialize`	11	2	11	9
`AnchorSerialize`	11	2	11	9
`associated_token`	3		3	3
`Box`	7	2	7	6
`bump`	17	2	17	16
`Bumps`	1			
`bumps`	16		17	17
`close`	7	2	6	6
`#[constant]`	4		5	3
`constraint`	8	2	7	10
`Context`	53	2	53	65
`CpiContext`	9	2	8	9
`declare_program!`				2
`#[derive(Accounts)]`	52		53	66
`emit!`	5	1	5	4
`emit_cpi!`	1		1	2
`err!`	4	1	3	3
`error!`	3	1	2	1
`#[error_code]`	7	2	6	7
`#[event]`	7	1	7	6
`#[event_cpi]`	1		1	1
`executable`	2		2	3
`has_one`	12	2	11	9
`init`	29	2	29	32
`init_if_needed`	3	1	3	4
`InitSpace`	4		5	4
`#[instruction]`	9		9	12

Construct	0.31.1	0.32.1	0.32.3	1.2.2
`Interface`	4	2	4	4
`InterfaceAccount`	7	2	7	6
`memo`	2		2	2
`mint`	6		6	6
`Option`	7	2	7	6
`owner`	8	2	7	8
`payer`	29		30	32
`Program`	32	2	32	37
`#[program]`	53	2	53	66
`realloc`	3	2	2	3
`remaining_accounts`	3	2	2	3
`rent_exempt`	2		2	2
`require!`	2	1	1	1
`require_eq!`	4	1	3	4
`require_family`	6		6	7
`require_gt!`	1		1	1
`require_gte!`	1		1	1
`require_keys_eq!`	5	1	4	5
`require_keys_neq!`	1		1	1
`require_neq!`	1		1	1
`seeds`	18	2	18	18
`Signer`	34	2	34	43
`space`	27		28	30
`SystemAccount`	3		2	3
`Sysvar`	4	1	3	2
`token`	8		8	7
`token_2022`	2		2	2
`token_interface`	7		7	6
`UncheckedAccount`	10	1	9	11
`zero`	9		10	11
`zero_copy`	6		6	5
`#[zero_copy]`	3	1	3	2

C Benchmark Results by Program

One row per measured program, ordered by Anchor version and name. "Kind" is *ex* for the Anchor repository's examples, *test* for its test programs and *prod* for production programs. "Instr." is instructions with weights over instructions declared. Gross and net are program totals in compute units, with a dash where the corpus holds no successful call of the program's instructions; sizes are in bytes, and "tx" is the number of transactions compared in the program's proof.

Program	Anchor	Kind	Instr.	Gross	Net	Red. %	Stock	Tare	Tx
account_command	0.31.1	test	1/1	5,297	2,689	49.2	185,592	133,680	584
account_info	0.31.1	test	1/1	559	200	64.2	163,344	100,920	480
allow_missing_optionals	0.31.1	test	1/1	4,490	2,242	50.1	178,808	121,192	596
basic_0	0.31.1	ex	1/1	197	132	33.0	162,176	99,504	440
basic_1	0.31.1	ex	2/2	5,185	2,114	59.2	182,144	126,056	688
basic_2	0.31.1	ex	2/2	5,736	2,255	60.7	184,456	129,176	696
basic_4	0.31.1	ex	2/2	12,027	8,488	29.4	187,976	129,720	680
basic_5	0.31.1	ex	5/5	12,030	6,348	47.2	189,320	137,088	3,152
bench	0.31.1	test	87/87	442,975	224,490	49.3	1,066,792	1,252,168	16,920
callee	0.31.1	test	5/5	7,306	3,484	52.3	183,032	125,680	2,856
caller	0.31.1	test	6/6	13,133	9,085	30.8	185,128	141,328	776
cashiers_check	0.31.1	test	3/3	23,018	18,258	20.7	218,000	168,712	1,805
chat	0.31.1	test	3/3	12,365	8,113	34.4	201,224	155,768	828
composite	0.31.1	test	2/2	3,690	1,231	66.6	179,776	120,448	696
custom_discriminator	0.31.1	test	7/7	9,507	5,146	45.9	182,600	124,232	1,024
declare_id	0.31.1	test	1/1	197	132	33.0	162,184	99,504	440
docs	0.31.1	test	1/1	576	269	53.3	164,944	104,584	504
errors	0.31.1	test	10/26	6,740	2,618	61.2	198,080	131,952	1,536
escrow	0.31.1	test	3/3	35,242	29,417	16.5	245,312	178,712	2,192
events	0.31.1	test	3/3	10,172	8,602	15.4	172,568	120,360	536
external	0.31.1	test	8/8	21,915	16,383	25.2	192,632	138,552	1,212
external	0.31.1	test	1/1	196	131	33.2	162,040	99,488	440
floats	0.31.1	test	2/2	6,028	2,813	53.3	185,752	133,936	704
generics	0.31.1	test	1/1	1,920	1,441	24.9	172,360	119,784	536
idl	0.31.1	test	3/4	30,661	20,465	33.3	255,656	221,344	1,440
idl_commands_one	0.31.1	test	1/1	197	132	33.0	162,200	99,512	440
idl_commands_two	0.31.1	test	1/1	201	133	33.8	162,240	99,504	440
ignore_non_accounts	0.31.1	test	1/1	810	259	68.0	163,744	101,120	480
init_if_needed	0.31.1	test	3/3	10,355	4,161	59.8	192,688	147,448	864
lamports	0.31.1	test	2/2	13,294	10,338	22.2	206,952	175,072	680
marinade_finance	0.31.1	prod	6/29	72,737	65,473	10.0	964,616	782,112	67
metadata	0.31.1	test	0/0	–	–	–	161,928	99,368	416
misc	0.31.1	test	64/65	451,661	349,325	22.7	716,656	738,288	9,731
misc_optional	0.31.1	test	63/63	382,117	296,233	22.5	767,912	812,984	11,510
multiple_suites	0.31.1	test	1/1	201	132	34.3	162,376	99,648	444
multiple_suites_run_single	0.31.1	test	1/1	197	132	33.0	162,216	99,520	440
native_system	0.31.1	test	12/12	12,941	4,898	62.2	196,152	124,832	1,568
optional	0.31.1	test	4/4	20,302	12,013	40.8	235,360	228,664	3,356
overflow_checks	0.31.1	test	4/4	10,936	3,581	67.3	189,984	140,256	2,084
puppet	0.31.1	ex	2/2	5,181	2,075	59.9	181,800	125,784	676
puppet_master	0.31.1	ex	1/1	3,492	2,211	36.7	175,144	116,264	520
pyth	0.31.1	test	0/2	–	–	–	166,552	108,776	576
realloc	0.31.1	test	3/3	18,244	13,085	28.3	210,376	187,456	1,544
relations_derivation	0.31.1	test	3/3	16,622	12,771	23.2	193,888	134,280	1,376
relations_derivation	0.31.1	test	2/2	12,129	8,900	26.6	188,952	129,272	700
remaining_accounts	0.31.1	test	3/3	10,538	4,570	56.6	198,504	144,032	880
safety_checks	0.31.1	test	1/1	504	200	60.3	162,840	100,928	480
spl_associated_token	0.31.1	test	0/0	–	–	–	162,096	99,392	416
spl_token	0.31.1	test	0/0	–	–	–	162,072	99,384	416
system_accounts	0.31.1	test	1/1	809	271	66.5	163,616	102,584	488
sysvars	0.31.1	test	1/1	1,132	509	55.0	167,488	110,424	480
token_proxy	0.31.1	test	8/8	71,905	62,135	13.6	300,744	230,264	1,824
token_wrapper	0.31.1	test	3/3	73,714	65,410	11.3	292,384	208,784	2,760
transfer_hook	0.31.1	test	0/2	–	–	–	230,264	200,320	852
typescript	0.31.1	test	1/1	197	132	33.0	162,184	99,504	440
validator_clone	0.31.1	test	1/1	197	132	33.0	162,192	99,512	440
zero_copy	0.31.1	test	7/7	21,301	14,367	32.6	207,528	154,328	3,848
zero_cpi	0.31.1	test	1/1	9,926	7,861	20.8	179,200	130,904	560
squads_multisig_program	0.32.1	prod	34/36	307,612	278,716	9.4	791,768	777,488	485
whirlpool	0.32.1	prod	20/65	184,638	173,399	6.1	1,556,816	1,445,224	217
account_command	0.32.3	test	1/1	5,150	2,689	47.8	185,080	133,680	584
account_info	0.32.3	test	1/1	559	200	64.2	162,704	100,920	480

Program	Anchor	Kind	Instr.	Gross	Net	Red. %	Stock	Tare	Tx
allow_missing_optionals	0.32.3	test	1/1	4,343	2,242	48.4	178,296	121,192	596
basic_0	0.32.3	ex	1/1	197	132	33.0	161,536	99,504	440
basic_1	0.32.3	ex	2/2	5,038	2,114	58.0	181,632	126,056	688
basic_2	0.32.3	ex	2/2	5,589	2,255	59.7	183,944	129,176	696
basic_4	0.32.3	ex	2/2	11,880	8,488	28.6	187,464	129,720	680
basic_5	0.32.3	ex	5/5	11,883	6,348	46.6	188,808	137,088	3,152
bench	0.32.3	test	87/87	429,745	224,490	47.8	1,066,280	1,252,168	16,920
callee	0.32.3	test	5/5	7,151	3,484	51.3	182,504	125,680	2,856
caller	0.32.3	test	6/6	12,722	9,079	28.6	184,456	141,328	776
cashiers_check	0.32.3	test	3/3	22,529	18,258	19.0	217,336	168,688	1,805
chat	0.32.3	test	3/3	12,218	8,113	33.6	200,712	155,768	828
composite	0.32.3	test	2/2	3,690	1,231	66.6	179,136	120,448	696
custom_discriminator	0.32.3	test	7/7	9,360	5,146	45.0	182,088	124,232	1,024
declare_id	0.32.3	test	1/1	197	132	33.0	161,544	99,504	440
docs	0.32.3	test	1/1	576	269	53.3	164,304	104,584	504
errors	0.32.3	test	10/26	6,740	2,618	61.2	197,424	131,952	1,536
escrow	0.32.3	test	3/3	34,297	29,420	14.2	244,848	178,760	2,192
events	0.32.3	test	3/3	10,041	8,602	14.3	171,928	120,360	536
external	0.32.3	test	8/8	21,766	16,383	24.7	192,104	138,552	1,212
external	0.32.3	test	1/1	196	131	33.2	161,400	99,488	440
floats	0.32.3	test	2/2	5,881	2,813	52.2	185,240	133,936	704
generics	0.32.3	test	1/1	1,920	1,441	24.9	171,720	119,784	536
idl	0.32.3	test	3/4	30,218	20,465	32.3	255,128	221,344	1,440
idl_commands_one	0.32.3	test	1/1	197	132	33.0	161,560	99,512	440
idl_commands_two	0.32.3	test	1/1	201	133	33.8	161,600	99,504	440
ignore_non_accounts	0.32.3	test	1/1	810	259	68.0	163,104	101,120	480
init_if_needed	0.32.3	test	3/3	10,060	4,161	58.6	192,176	147,448	864
lamports	0.32.3	test	2/2	13,000	10,338	20.5	206,440	175,072	680
metadata	0.32.3	test	0/0	–	–	–	161,288	99,368	416
misc	0.32.3	test	64/65	444,318	349,341	21.4	716,376	738,520	9,731
misc_optional	0.32.3	test	63/63	376,515	296,243	21.3	767,632	813,216	11,510
multiple_suites	0.32.3	test	1/1	201	132	34.3	161,736	99,648	444
multiple_suites_run_single	0.32.3	test	1/1	197	132	33.0	161,576	99,520	440
native_system	0.32.3	test	12/12	12,941	4,898	62.2	195,512	124,832	1,568
new_idl	0.32.3	test	13/13	35,181	18,759	46.7	283,536	249,768	1,880
optional	0.32.3	test	4/4	19,858	12,013	39.5	234,832	228,664	3,356
overflow_checks	0.32.3	test	4/4	10,789	3,581	66.8	189,472	140,256	2,084
puppet	0.32.3	ex	2/2	5,034	2,075	58.8	181,288	125,784	676
puppet_master	0.32.3	ex	1/1	3,361	2,211	34.2	174,504	116,264	520
pyth	0.32.3	test	0/2	–	–	–	165,912	108,776	576
realloc	0.32.3	test	3/3	17,966	13,085	27.2	209,840	187,456	1,544
relations_derivation	0.32.3	test	2/2	11,982	8,900	25.7	188,440	129,272	700
relations_derivation	0.32.3	test	3/3	16,475	12,771	22.5	193,376	134,280	1,376
remaining_accounts	0.32.3	test	3/3	10,244	4,570	55.4	197,992	144,032	880
safety_checks	0.32.3	test	1/1	504	200	60.3	162,200	100,928	480
spl_associated_token	0.32.3	test	0/0	–	–	–	161,456	99,392	416
spl_token	0.32.3	test	0/0	–	–	–	161,432	99,384	416
system_accounts	0.32.3	test	1/1	809	271	66.5	162,976	102,584	488
sysvars	0.32.3	test	1/1	1,132	509	55.0	166,848	110,424	480
token_proxy	0.32.3	test	8/8	70,312	62,140	11.6	300,464	230,496	1,824
token_wrapper	0.32.3	test	3/3	71,828	65,416	8.9	292,104	209,016	2,760
transfer_hook	0.32.3	test	0/2	–	–	–	229,752	200,320	852
typescript	0.32.3	test	1/1	197	132	33.0	161,544	99,504	440
validator_clone	0.32.3	test	1/1	197	132	33.0	161,552	99,512	440
zero_copy	0.32.3	test	7/7	21,154	14,367	32.1	207,016	154,328	3,848
zero_cpi	0.32.3	test	1/1	9,763	7,861	19.5	178,560	130,904	560
account_command	1.2.2	test	1/1	5,200	2,749	47.1	129,808	87,992	176
account_generation_test	1.2.2	test	1/1	4,371	1,886	56.9	116,576	54,264	164
account_info	1.2.2	test	1/1	581	198	65.9	56,128	35,352	72
accountloader_realloc	1.2.2	test	6/6	29,425	21,409	27.2	161,768	132,296	1,836
allow_missing_optionals	1.2.2	test	1/1	4,420	2,277	48.5	117,824	59,768	188
anchor_cli_legacy_idl	1.2.2	test	1/1	538	212	60.6	167,752	84,960	488
basic_0	1.2.2	ex	1/1	185	128	30.8	54,992	33,808	32
basic_1	1.2.2	ex	2/2	5,686	2,159	62.0	129,736	63,832	280
basic_2	1.2.2	ex	2/2	6,267	2,352	62.5	132,368	68,816	288
basic_4	1.2.2	ex	2/2	12,499	8,611	31.1	135,720	70,904	272
basic_5	1.2.2	ex	5/5	14,556	6,585	54.8	138,400	78,768	1,520
bench	1.2.2	test	87/87	438,316	227,501	48.1	833,576	1,193,784	16,512
callee	1.2.2	test	7/7	8,221	4,030	51.0	122,584	64,560	628
caller	1.2.2	test	10/10	39,739	30,835	22.4	137,840	119,952	816
cashiers_check	1.2.2	test	3/3	26,269	18,195	30.7	152,184	126,376	989
chat	1.2.2	test	3/3	12,315	8,057	34.6	143,872	102,416	420
composite	1.2.2	test	2/2	7,436	1,874	74.8	122,760	65,464	288
custom_discriminator	1.2.2	test	7/7	9,541	5,123	46.3	121,472	62,048	616
custom_program	1.2.2	test	1/1	21,479	20,734	3.5	72,472	58,008	72
declare_id	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32

Program	Anchor	Kind	Instr.	Gross	Net	Red. %	Stock	Tare	Tx
declare_program	1.2.2	test	3/4	36,744	31,668	13.8	134,176	100,416	472
docs	1.2.2	test	1/1	596	248	58.4	78,840	39,200	96
duplicate_mutable_accounts	1.2.2	test	9/9	32,345	12,354	61.8	182,288	159,704	1,384
env_id	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32
errors	1.2.2	test	10/26	7,585	2,683	64.6	126,984	72,128	1,128
escrow	1.2.2	test	3/3	40,025	29,933	25.2	195,656	137,184	1,376
events	1.2.2	test	3/3	3,824	2,588	32.3	91,352	70,032	120
example	1.2.2	test	1/1	288	252	12.5	55,072	37,424	32
external	1.2.2	test	13/13	37,968	25,665	32.4	172,560	113,408	1,340
external	1.2.2	test	1/1	581	245	57.8	61,032	38,968	96
floats	1.2.2	test	2/2	6,561	2,841	56.7	133,872	89,256	296
generics	1.2.2	test	1/1	2,016	1,439	28.6	89,008	68,296	128
idl_commands_one	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32
idl_commands_two	1.2.2	test	1/1	187	128	31.6	54,992	33,808	32
ignore_non_accounts	1.2.2	test	1/1	869	258	70.3	56,520	35,504	72
init_if_needed	1.2.2	test	3/3	11,325	4,278	62.2	145,760	85,152	456
lamports	1.2.2	test	2/2	12,955	10,280	20.6	149,376	129,104	272
malicious	1.2.2	test	1/1	666	309	53.6	56,016	35,696	80
matching_solana_program	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32
metadata	1.2.2	test	0/0	–	–	–	54,768	33,808	8
misc	1.2.2	test	64/65	454,635	348,362	23.4	636,304	701,992	9,323
misc_optional	1.2.2	test	63/63	386,240	294,633	23.7	695,384	772,760	11,102
mismatched_solana_program	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32
multiple_suites	1.2.2	test	1/1	189	129	31.7	72,608	33,968	36
multiple_suites_run_single	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32
native_system	1.2.2	test	12/12	13,833	4,927	64.4	113,856	72,744	1,160
new	1.2.2	test	2/2	8,977	3,878	56.8	128,272	69,496	320
old	1.2.2	test	1/1	4,400	1,948	55.7	117,208	55,480	164
optional	1.2.2	test	4/4	23,103	12,222	47.1	186,048	186,288	2,132
overflow_checks	1.2.2	test	4/4	12,698	3,731	70.6	139,880	86,144	860
puppet	1.2.2	ex	2/2	5,679	2,149	62.2	129,408	63,560	268
puppet_master	1.2.2	ex	1/1	4,415	3,332	24.5	99,936	52,008	112
pyth	1.2.2	test	0/2	–	–	–	78,960	41,976	168
realloc	1.2.2	test	6/6	35,996	23,750	34.0	188,744	183,008	2,276
relations_derivation	1.2.2	test	2/2	11,993	8,934	25.5	127,928	71,816	292
relations_derivation	1.2.2	test	3/3	16,823	12,901	23.3	133,640	77,720	560
remaining_accounts	1.2.2	test	3/3	10,418	4,756	54.3	135,288	83,960	472
spl_associated_token	1.2.2	test	0/0	–	–	–	54,784	33,824	8
spl_token	1.2.2	test	0/0	–	–	–	54,784	33,824	8
system_accounts	1.2.2	test	1/1	870	270	69.0	56,608	36,960	80
sysvars	1.2.2	test	1/1	1,228	512	58.3	60,856	44,280	72
tare_registry	1.2.2	prod	4/4	84,919	65,361	23.0	281,992	278,960	17
test_instruction_validation	1.2.2	test	2/2	1,724	1,047	39.3	75,016	40,384	168
test_instruction_validation	1.2.2	test	3/3	2,667	1,653	38.0	76,272	41,184	248
tictactoe	1.2.2	test	5/5	12,185	3,863	68.3	123,816	85,488	1,240
token_proxy	1.2.2	test	8/8	75,485	62,708	16.9	244,520	183,896	1,416
token_wrapper	1.2.2	test	3/3	79,250	65,962	16.8	236,232	167,864	1,944
transfer_hook	1.2.2	test	0/2	–	–	–	170,992	142,224	444
typescript	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32
unchecked_account	1.2.2	test	1/1	526	198	62.4	55,616	35,352	72
validator_clone	1.2.2	test	1/1	185	128	30.8	54,992	33,808	32