Drips-Network Security Review

Auditors

Saw-mon and Natalie, Lead Security Researcher

Optimum, Lead Security Researcher

Rusty Rabbit, Security Researcher

Rappie, Junior Security Researcher

Report prepared by: Lucas Goiriz

November 7, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Drips is a protocol and app built on Ethereum that enables organizations and individuals to directly and publicly provide funding to the free and open source software projects they depend on the most.

Drips also includes gas-optimized and integrated primitives for streaming and splitting tokens, allowing users and web3 apps to stream and split funds by the second with continuous settlement for use cases like contributor payments, vesting and subscription memberships.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of commit afeba5...e28fa0 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Drips engaged with Spearbit to review the contracts protocol.

The contracts in scope exist in Ethereum mainnet under the following addresses:

Drips	0xd0Dd053392db676D57317CD4fe96Fc2cCf42D0b4
Drips logic	0xb0C9B6D67608bE300398d0e4FB0cCa3891E1B33F
Caller	0x60F25ac5F289Dc7F640f948521d486C964A248e5
AddressDriver	0x1455d9bD6B98f95dd8FEB2b3D60ed825fcef0610
AddressDriver logic	0x3Ea1e774f98cc4C6359bbCB3238E3e60365Fa5c9
NFTDriver	0xcf9c49B0962EDb01Cdaa5326299ba85D72405258
NFTDriver logic	0x3B11537D0d4276Ba9e41FFe04e9034280bd7af50
ImmutableSplitsDriver	0x1212975c0642B07F696080ec1916998441c2b774
ImmutableSplitsDriver logic	0x2c338CDf00dFd5A9B3B6b0b78BB95352079AAF71
RepoDriver	0x770023d55D09A9C110694827F1a6B32D5c2b373E
RepoDriver logic	0xfC446dB5E1255e837E95dB90c818C6fEb8e93ab0

The verification was performed independently by Spearbit through the same deployment environment, kindly provided by the Drips team, to generate the runtime bytecode of the contracts and comparing it with the existing runtime bytecode in mainnet, fetch via the Ethereum JSON-RPC API.

In this period of time a total of **29** issues were found.

Summary

Project Name	Drips
Repository	contracts
Commit	afeba55f
Type of Project	Crowdfunding, DeFi
Audit Timeline	Sep 13 - Sep 27
Two week fix period	Sep 27 - Oct 11

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	5	0	5
Gas Optimizations	8	0	8
Informational	16	0	16
Total	**29**	**0**	**29**

5 Findings

5.1 Low Risk

5.1.1 Pausers can unpause the protocol

Severity: Low Risk

Context: Managed.sol#L170-L173

Description: As pausers are usually systems that can react quickly to do privileged operations on the protocol, the important one being pausing the protocol, the `unpause()` operation should be only performed by the admins since the pausers might not have as secure of a setup as the admins.

Recommendation: It would be best to restrict the `unpause()` operation to only the admins of the `Managed` contracts.

Drips Network: The `pauser` privilege can be given not only to an automated system, but also to other entities, e.g. a multisig without a timelock. In that case it makes sense to be able to unpause as easily as to pause, so the protocol is frozen for as short period as possible. Having it blocked for the minimum of 2 days makes each pause a major disruption that should be avoided at all costs, which hurts the security in case the danger isn't obviously critical.

The ability to unpause can be easily removed by introducing an intermediate contract controlled by the monitoring tool, that can only call `pause`. It can even provide more sophistication, e.g. a cooldown period or an aggregator of signals from multiple tools. Adding such sophistication to each pausable contract is probably an overkill, with the possibility of intermediate contracts in mind, a simple "pauser" role should be enough.

Spearbit: Acknowledged.

5.1.2 No pausers are set for `Drips` and its drivers on the current live contracts

Severity: Low Risk

Context: Managed.sol#L154-L156

Description: No pausers are set for any of the drivers or the `Drips` contract. All these contracts share the same `admin` which is a `Timelock` contract with the delay of `2 days`. So in case the protocol needs to be paused, it would take at least `2 days` to pause it.

Recommendation: Although this is not a smart-contract vulnerability, it is considered an operational security risk.

It would be best to assign a `pauser` to these contracts immediately before mainstream adoption so that the protocol can be paused in an event of a live vulnerability.

This an important issue that would need to be addressed immediately as there would need to be voting campaigns to assign the `pauser`.

Drips Network: This is correct, and this is why the `pauser` role even exists. Because of it we can introduce something more robust than a 2 day publicly announced voting campaign, for example an automated monitoring tool or a small multisig.

Acknowledged, we're working on this.

Spearbit: Acknowledged.

5.1.3 `uint32` type parameter for timestamps might not suffice in the long run

Severity: Low Risk

Context: Global scope

Description: Time parameters used in this codebase have the type `uint32`. The current timestamp fits into `31` bits and so in about `80` years the timestamp would not fit into `uint32`.

Note that `geth` uses `uint64` for timestamps and the execution spec uses `uint256`.

Recommendation: It might be best to use `uint64` for time parameters to avoid the above issue.

Drips Network: Acknowledged, but won't be fixed. The timestamps are stored on-chain, and increasing their size would prevent some variables from fitting in storage slots leading to an increased gas usage.

Spearbit: Acknowledged.

5.1.4 Users can front run calls to `Drips.split` and change the receivers list

Severity: Low Risk

Context: Splits.sol#L152, Splits.sol#L228

Description: The splits feature allows users with `balance.splittable > 0` to split received streams and allocated portions of it to a pre-determined list of receivers. Users can also set the list of receivers at any moment without any restrictions whatsoever, which may lead to potential front running attacks where the balance owner calls `Drips.setSplits` right before a call to `Drips.split` and, by doingt this, steal the balance that was supposed to be allocated to the original receivers.

Balance owners are trusted by the recipients to keep the original receivers list or at least only change the receivers list after the balance was distributed.

Recommendation: The team is well aware of this issue/trust assumption. During the audit we discussed a potential partial mitigation that includes splitting the current balance as part of `Drips.setSplits` before applying the actual change to the receivers list, but it requires major changes in the protocol that will also add additional gas costs.

Drips Network: Acknowledged. We will inform the users about this trust assumption.

Spearbit: Acknowledged.

5.1.5 `NFTDriver`: NFT driver sellers might trick potential buyers to buy a depreciated token on NFT marketplaces

Severity: Low Risk

Context: NFTDriver.sol#L21

Description: The `NFTDriver` contract is an ERC721 token, representing ownership over an NFT driver that is a method of authentication based on the holder of the NFT. These tokens are transferable and can be traded on decentralized exchanges. However, a malicious NFT driver seller can potentially front-run the actual DEX swap transaction in the mempool, causing the buyer to purchase a depreciated token.

The full scenario:

1. A potential seller is minting an NFT driver.
2. At some point after that, this NFT holds value (i.e. having `balance.splittable > 0`, `balance.collectable > 0`, or balance for streaming).
3. The NFT driver owner is now calling `approve` with the corresponding `tokenId` and an address of a DEX (and, if needed, signs a maker order), the price of this token will be probably based on the value that it holds.
4. A potential buyer now calls the function that does the atomic swap. The transaction is still in the mempool.

5. At this point the malicious seller can call any of the functions that will result in the depreciation of the token value, i.e.: `collect`, `split`, `setStreams`.
6. The atomic swap transaction succeeds, the buyer gets a depreciated token but the seller gets the whole payment.

Recommendation: The issue is caused due to the fact that the `tokenId` is not tied to changes in the token storage state. For the general case, we propose implementing a mechanism where there are two different identifiers for any token: an internal identifier that should stay constant throughout the entire token life-cycle (it can be implemented as a counter), and an external identifier. The internal identifier will be used to find the corresponding storage data attached to the token.

Whenever the token storage data is changed (`collect`, `split`, `setStreams` for example), the old token should be burned, and a new token should be minted using a new external identifier. The external identifier might be implemented as `hash(internal_id, version)`, where the version is incremented whenever the token storage data is changed.

If the team decided not to follow the recommendation above due to over-complexity or additional gas costs, at least consider informing users that NFT drivers are not meant to be tradeable in trustless marketplaces, and that potential buyers might be exposed to the risk described above.

Drips Network: That's a good idea.

Burning the NFTs would be costly and wouldn't solve the problem. (It would also clear the approvals, which would defeat their purpose.) `receiveStreams` and `split` are called directly on `Drips`, without notifying the driver, and this is exactly how the architecture is supposed to work: it would be a very different protocol if drivers were called back. Burning on each call to `setStreams` wouldn't be very beneficial either, it would facilitate transferring accounts with funds that are currently in the streamed balance, but that's an awkward way to sell tokens. `collect` is probably the only function that makes sense to trigger burning, but it too would just facilitate the sale of tokens. Overall, the only thing worth selling would be the position in the dependency graph and control over future, incoming funds. The tokens currently available for withdrawal can and should be sold differently.

Spearbit: Acknowledged.

5.2 Gas Optimization

5.2.1 Cheaper way of ensuring the requirements for the good case in `_assertSplitsValid(...)`

Severity: Gas Optimization

Context: Splits.sol#L251-L254

Description/Recommendation: In the `_assertSplitsValid(...)` loop we have the following `require` statements:

```
require(weight != 0, "Splits receiver weight is zero");
// ...
if (i > 0) require(prevAccountId < accountId, "Splits receivers not sorted");
```

To favour a more gas efficient setup for the good case, one can instead have an accumulator flag (just one bit is enough) and check after the loop if the flag is set and and then throw. In each iteration you can `OR` the flag with the conditions for that iteration.

Similar techniques can be applied to other loops where requirements are checked.

Drips Network: I've switched to (a rather verbose and inelegant):

```
// ...
bool foundZeroWeight = false;
// ...
foundZeroWeight = foundZeroWeight || weight == 0;
// ...
require(foundZeroWeight == false, "Splits receiver weight is zero");
// ...
```

The gas usage increased in some tests and stayed the same in others, overall:

```
Overall gas change: 6651 (0.002%)
```

Spearbit: Acknowledged.

5.2.2 `_assertSplitsValid(...)` can be optimised

Severity: Gas Optimization

Context: Splits.sol#L254

Description/Recommendation: Depending on if we would like to allow `accountId` to be `0` or not (in the case of `0` `accountId` that would mean the driver ID and also the sub account ID would need to be `0`. With the current deployed contracts, that would be the `AddressDriver` and that means we are setting the receiver to be the `address(0)`), then the following line

```
require(prevAccountId < accountId, "Splits receivers not sorted");
```

can be changed to

```
- if (i > 0) require(prevAccountId < accountId, "Splits receivers not sorted");
+ require(prevAccountId < accountId, "Splits receivers not sorted");
```

and with some corrections to the test files:

```
  diff --git a/test/Splits.t.sol b/test/Splits.t.sol
  index 98de14e..b664812 100644
- a/test/Splits.t.sol
+ b/test/Splits.t.sol
  @@ -161,7 +161,7 @@ contract SplitsTest is Test, Splits {
           SplitsReceiver[] memory receiversGood = new SplitsReceiver[](countMax);
           SplitsReceiver[] memory receiversBad = new SplitsReceiver[](countMax + 1);
           for (uint256 i = 0; i < countMax; i++) {
-              receiversGood[i] = SplitsReceiver(i, 1);
+              receiversGood[i] = SplitsReceiver(i+1, 1);
               receiversBad[i] = receiversGood[i];
           }
           receiversBad[countMax] = SplitsReceiver(countMax, 1);
  @@ -317,6 +317,12 @@ contract SplitsTest is Test, Splits {
           uint256 receiversLengthRaw,
           uint256 totalWeightRaw
       ) internal view returns (SplitsReceiver[] memory receivers) {
+        for (uint256 i = 0; i < receiversRaw.length; i++) {
+            if(receiversRaw[i].accountId == 0) {
+                ++receiversRaw[i].accountId;
+            }
+        }
+
           for (uint256 i = 0; i < receiversRaw.length; i++) {
               for (uint256 j = i + 1; j < receiversRaw.length; j++) {
                   if (receiversRaw[i].accountId > receiversRaw[j].accountId) {
```

we would have `forge snapshot --diff .gas-non-zero-accountId`:

```
testSplitsConfigurationIsCommonBetweenTokens() (gas: 2 (0.001%))
testSplitSplitsFundsReceivedFromAllSources() (gas: 6 (0.001%))
testUncollectedFundsAreSplitUsingCurrentConfig() (gas: 6 (0.001%))
testSimpleSplit() (gas: 3 (0.002%))
testForwardSplits() (gas: 5 (0.002%))
testSplitRevertsIfInvalidCurrSplitsReceivers() (gas: 3 (0.003%))
testSetSplits() (gas: 3 (0.005%))
testSetSplits() (gas: 3 (0.005%))
testSetSplits() (gas: 3 (0.005%))
testAccountCanSplitToItself() (gas: -13 (-0.005%))
testSplitMultipleReceivers() (gas: -13 (-0.005%))
testRejectsTooHighTotalWeightSplitsReceivers() (gas: 6 (0.005%))
testSplittingSplitsAllFundsEvenWhenTheyDoNotDivideEvenly() (gas: -16 (-0.006%))
testCanSplitAllWhenCollectedDoesNotSplitEvenly() (gas: -16 (-0.007%))
testSplitFundsAddUp(uint256,address,uint128,(uint256,uint32)[200],uint256,uint256) (gas: -1061
↪  (-0.015%))
testCreateSplits() (gas: -16 (-0.016%))
testLimitsTheTotalSplitsReceiversCount() (gas: -3778 (-0.039%))
testRejectsUnsortedSplitsReceivers() (gas: -16 (-0.084%))
testRejectsDuplicateSplitsReceivers() (gas: -16 (-0.094%))
Overall gas change: -4905 (-0.002%)
```

If we keep the driver ID for the `AddressDriver` `0`, the recommended changes would only mean that one can not set a split receiver to `address(0)`.

Drips Network: You're right, and it would save some gas on the `if` part. OTOH it would make the API more hairy, we'd need to warn everywhere that attempting to support account `0` will throw. It would be elegant to make the driver IDs go from `1` instead of `0` to ensure that account ID `0` is indeed guaranteed to be invalid, but then the `AddressDriver` user IDs would no longer be equal to the addresses themselves. I think that performing 1 `if` per receiver is a worthy trade off.

I would still prefer to avoid , IMO the gas saving is not worth introducing an API special case and a hard requirement on the behavior of the driver that's registered under ID 0.

But there is an elegant solution:

```
if(accountId <= prevAccountId) require(i == 0, "Splits receivers not sorted");
```

This way the happy path is almost as cheap as what you've proposed without introducing any new API restrictions:

```
testSplitsConfigurationIsCommonBetweenTokens() (gas: 1 (0.000%))
testSplitSplitsFundsReceivedFromAllSources() (gas: 4 (0.001%))
testUncollectedFundsAreSplitUsingCurrentConfig() (gas: 4 (0.001%))
testSplit() (gas: 2 (0.001%))
testSetSplitsTrustsForwarder() (gas: 2 (0.001%))
testSetSplitsTrustsForwarder() (gas: 2 (0.001%))
testSetSplitsTrustsForwarder() (gas: 2 (0.001%))
testForwardSplits() (gas: 3 (0.001%))
testSetSplits() (gas: 2 (0.001%))
testSetSplits() (gas: 2 (0.001%))
testSetSplits() (gas: 2 (0.001%))
testSimpleSplit() (gas: 2 (0.001%))
testSplitRevertsIfInvalidCurrSplitsReceivers() (gas: 2 (0.002%))
testSetSplits() (gas: 2 (0.003%))
testSetSplits() (gas: 2 (0.003%))
testSetSplits() (gas: 2 (0.003%))
testRejectsTooHighTotalWeightSplitsReceivers() (gas: 4 (0.004%))
testAccountCanSplitToItself() (gas: -14 (-0.005%))
testSplitMultipleReceivers() (gas: -14 (-0.005%))
testSplittingSplitsAllFundsEvenWhenTheyDoNotDivideEvenly() (gas: -17 (-0.007%))
testCanSplitAllWhenCollectedDoesNotSplitEvenly() (gas: -17 (-0.007%))
testRejectsZeroWeightSplitsReceivers() (gas: -1 (-0.007%))
testRejectsUnsortedSplitsReceivers() (gas: 2 (0.010%))
testRejectsDuplicateSplitsReceivers() (gas: 2 (0.012%))
testSplitFundsAddUp(uint256,address,uint128,(uint256,uint32)[200],uint256,uint256) (gas: -948 (-0.015%))
testCreateSplits() (gas: -17 (-0.017%))
testLimitsTheTotalSplitsReceiversCount() (gas: -3760 (-0.039%))
Overall gas change: -4744 (-0.002%)
```

This will be fixed in a future release.

Spearbit: Acknowledged.

5.2.3 `totalWeight` in `_assertSplitsValid(...)` can have a `uint256` type

Severity: Gas Optimization

Context: Splits.sol#L245

Description/Recommendation: `uint64` is chosen for `totalWeight` so that if the max amount of receivers are used `200` and all the weights are `type(uint32).max` it would still fit into `uint64` for later comparison. One can instead use `uint256` to avoid cleanups per iteration.

```
uint256 totalWeight = 0;
```

The output of `forge snapshot --diff` is the following:

```
testLimitsTheTotalSplitsReceiversCount() (gas: -6 (-0.000%))
testSplitFundsAddUp(uint256,address,uint128,(uint256,uint32)[200],uint256,uint256) (gas: -6 (-0.000%))
testSplitSplitsFundsReceivedFromAllSources() (gas: -12 (-0.002%))
testUncollectedFundsAreSplitUsingCurrentConfig() (gas: -12 (-0.002%))
testSplitsConfigurationIsCommonBetweenTokens() (gas: -5 (-0.002%))
testAccountCanSplitToItself() (gas: -5 (-0.002%))
testSplitMultipleReceivers() (gas: -5 (-0.002%))
testSplittingSplitsAllFundsEvenWhenTheyDoNotDivideEvenly() (gas: -6 (-0.002%))
testCanSplitAllWhenCollectedDoesNotSplitEvenly() (gas: -6 (-0.002%))
testSimpleSplit() (gas: -6 (-0.004%))
testForwardSplits() (gas: -10 (-0.004%))
testSplitRevertsIfInvalidCurrSplitsReceivers() (gas: -6 (-0.006%))
testCreateSplits() (gas: -6 (-0.006%))
testSetSplits() (gas: -6 (-0.009%))
testSetSplits() (gas: -6 (-0.009%))
testSetSplits() (gas: -6 (-0.009%))
testRejectsTooHighTotalWeightSplitsReceivers() (gas: -12 (-0.011%))
Overall gas change: -121 (-0.000%)
```

This issue is related to issue "`SplitsReceiver.weights` can be packed into a smaller type".

Drips Network: Will be fixed in a future release.

Spearbit: Acknowledged.

5.2.4 Simpler and cheaper way of calculating `currSplitAmt` and `splitAmt` in `_split(...)`

Severity: Gas Optimization

Context: Splits.sol#L169-L171

Description/Recommendation: A simpler and cheaper way of calculating `currSplitAmt` and `splitAmt` in `_split(...)` would be the following:

```
uint128 currSplitAmt = splitAmt;
splitAmt = uint128(collectableAmt * splitsWeight / _TOTAL_SPLITS_WEIGHT);
currSplitAmt = splitAmt - currSplitAmt;
```

The output of `forge snapshot --diff` is the following:

```
testUncollectedFundsAreSplitUsingCurrentConfig() (gas: -12 (-0.002%))
testSplitSplitsFundsReceivedFromAllSources() (gas: -24 (-0.003%))
testSplitsConfigurationIsCommonBetweenTokens() (gas: -19 (-0.007%))
testForwardSplits() (gas: -19 (-0.007%))
testSimpleSplit() (gas: -12 (-0.007%))
testSplitMultipleReceivers() (gas: -20 (-0.008%))
testSplittingSplitsAllFundsEvenWhenTheyDoNotDivideEvenly() (gas: -24 (-0.009%))
testSplitFundsAddUp(uint256,address,uint128,(uint256,uint32)[200],uint256,uint256) (gas: -624 (-0.010%))
testCanSplitAllWhenCollectedDoesNotSplitEvenly() (gas: -24 (-0.010%))
testAccountCanSplitToItself() (gas: -39 (-0.014%))
Overall gas change: -817 (-0.000%)
```

Note that the formulas used for `currSplitAmt` (a_i) and `splitAmt` (A_i) are:

$$A_i = \left\lfloor \frac{a \sum_{0 \le j \le i} w_j}{10^6} \right\rfloor$$

and

$$a_i = A_i - A_{i-1} = \left\lfloor \frac{aw_i}{10^6} + \frac{a\sum_{0\le j\le i-1} w_j}{10^6} \right\rfloor$$

where a is the original splittable amount and w_i s are the weights. Depending on the position of a receiver in the list the split amount they receive might have `1 wei` favoured to them.

We can still run this loop backwards although different set of receivers would be favoured for receive potential `1` extra `wei`. See issue "Iterate loops backwards when possible".

Drips Network: Will be fixed in a future release.

Spearbit: Acknowledged.

5.2.5 Iterate loops backwards when possible

Severity: Gas Optimization

Context: Splits.sol#L134-L136

Description/Recommendation:

- Splits.sol#L134-L136 would be cheaper if we iterate backwards:

```
for (uint256 i = currReceivers.length; i != 0; ) {
    --i;
    splitsWeight += currReceivers[i].weight;
}
```

Check the output of `forge snapshot --diff` below:

```
testReceiveAllStreamsCycles() (gas: -2 (-0.000%))
testGiveLimitsTotalBalance() (gas: -3 (-0.001%))
testReceiveSomeStreamsCycles() (gas: -3 (-0.001%))
testStreamsInDifferentTokensAreIndependent() (gas: -10 (-0.001%))
testSqueezeStreams() (gas: -5 (-0.001%))
testFundsGivenFromAccountCanBeCollected() (gas: -2 (-0.001%))
testAccountCanSplitToItself() (gas: -5 (-0.002%))
testUncollectedFundsAreSplitUsingCurrentConfig() (gas: -17 (-0.002%))
testSplitSplitsFundsReceivedFromAllSources() (gas: -28 (-0.004%))
testSimpleSplit() (gas: -14 (-0.009%))
testForwardSplits() (gas: -25 (-0.009%))
testSplitsConfigurationIsCommonBetweenTokens() (gas: -27 (-0.010%))
testSplitMultipleReceivers() (gas: -25 (-0.010%))
testSplittingSplitsAllFundsEvenWhenTheyDoNotDivideEvenly() (gas: -31 (-0.012%))
testCanSplitAllWhenCollectedDoesNotSplitEvenly() (gas: -31 (-0.013%))
Overall gas change: -228 (-0.000%)
```

The above technique applies to other similar loops as well if formulas in the loop body are not dependant on the order of index iteration.

This issue is related to "`totalWeight` in `_assertSplitsValid(...)` can have a `uint256` type".

Drips Network: Will be fixed in a future release.

Spearbit: Acknowledged.

5.2.6 `splitsWeight` can be `uint256` to avoid cleanups per iteration

Severity: Gas Optimization

Context: Splits.sol#L166, Splits.sol#L133

Description/Recommendation: It would make sense to use `uint256` for `splitsWeight` to avoid the clean up cost per iteration and also making sure the multiplication of `amount * splitsWeight` does not overflow.

See the output of `forge snapshot --diff` (for only changing Splits.sol#L166) below:

```
testUncollectedFundsAreSplitUsingCurrentConfig() (gas: -18 (-0.002%))
testSplitSplitsFundsReceivedFromAllSources() (gas: -36 (-0.005%))
testSplitRevertsIfInvalidCurrSplitsReceivers() (gas: -36 (-0.033%))
testSplitFundsAddUp(uint256,address,uint128,(uint256,uint32)[200],uint256,uint256) (gas: -2892
↪  (-0.044%))
testSimpleSplit() (gas: -258 (-0.160%))
testGive() (gas: -60 (-0.187%))
testAccountCanSplitToItself() (gas: -605 (-0.219%))
testForwardSplits() (gas: -595 (-0.225%))
testSplittingSplitsAllFundsEvenWhenTheyDoNotDivideEvenly() (gas: -588 (-0.226%))
testSplitMultipleReceivers() (gas: -596 (-0.229%))
testCanSplitAllWhenCollectedDoesNotSplitEvenly() (gas: -588 (-0.245%))
testSplitsConfigurationIsCommonBetweenTokens() (gas: -816 (-0.290%))
Overall gas change: -7088 (-0.003%)
```

Drips Network: Will be fixed in a future release.

Spearbit: Acknowledged.

5.2.7 Pre-increment in `Caller`'s `unauthorizeAll()`

Severity: Gas Optimization

Context: Caller.sol#L153

Description/Recommendation: The following operation

```
++_authorized[sender].clears;
```

is cheaper according to `forge snapshot --diff`:

```
testUnauthorizeAllUnauthorizesAll() (gas: -2 (-0.001%))
testCallerCanCallOnItselfUnuthorizeAll() (gas: -2 (-0.002%))
Overall gas change: -4 (-0.000%)
```

Drips Network: Won't fix, but probably Solidity will fix that inefficiency for a future release. It's just 2 gas saved on a rarely used function at the cost of the slightly decreased code readability. I've opened an issue, I'm very surprised than people have spent countless hours working around this clear optimizer deficiency and nobody opened an issue in Solidity for that . We acknowledge the issue.

Spearbit: Acknowledged.

5.2.8 `Splits._setSplits` : The `SplitsSet` event will be emitted even in a no-op transaction

Severity: Gas Optimization

Context: Splits.sol#L231

Description: In case `_setSplits` is being called with the same receivers list, the `SplitsSet` event will be emitted although the transaction will end up in a no-op.

Recommendation: Consider reverting the transaction for this no-op case or at the least move the event emission into the `if` block instead.

Drips Network: I would prefer avoiding a revert for the no-op case becausse it makes the API more complicated to use with the extra edge case, which isn't really needed for anything. Ideally the caller should always check the current configuration against the applied configuration hash, this would actually increase the total gas usage.

The event emission will be moved into the `if` block in the next release.

Spearbit: Acknowledged.

5.3 Informational

5.3.1 `SplitsReceiver.weights` can be packed into a smaller type

Severity: Informational

Context: Splits.sol#L13

Description/Recommendation: `SplitsReceiver.weights` can be packed into a smaller type.`_TOTAL_SPLITS_WEIGHT = '0b11110100001001000000'` fits into `20` bits. So we could have used `uint24` for `SplitsReceiver.weights`. This issue is related to "`totalWeight` in `_assertSplitsValid(...)` can have a `uint256` type".

In case of expanding the type to `uint256` all the `unchecked` blocks would need to be analyzed again to make sure the arithmetic would not over/underflow.

Drips Network: That's a good point, it makes no sense. It's a leftover from the older version of the contract, but now it could be `uint24`. If I were to change it, I'd probably make it `uint256` to neatly fill calldata and save on its verification on access. While it's a nice catch, we're not going to fix it in this version of the protocol. It's a breaking change to the API, that doesn't bring any benefits except a small clarification. We may fix it in a future version of the protocol.

Spearbit: Acknowledged.

5.3.2 `RepoDriver`'s `Forge` endpoints need to be monitored incase of an api change

Severity: Informational

Context: RepoDriver.sol#L354-L368

Description/Recommendation: The hardcoded api links needs monitoring to make sure the `GitLab` and `GitHub` api endpoint structures don't change over time.

Drips Network: Good point, acknowledged.

Spearbit: Acknowledged.

5.3.3 The subsidized `LINK` tokens for `RepoDriver` can be abused

Severity: Informational

Context: RepoDriver.sol#L261-L269

Description: Anyone can call the endpoint in this context `requestUpdateOwner(...)` to transfer all `RepoDriver`'s `LINK` token balance (cause all its balance to be transferred to the `Operator` and eventually the `Operator`'s owner).

Current balance is `1,410.665786792332952542 LINK`, around $ 9620.74.

Recommendation: No action is required but might be best to document and setup monitoring for abusive behaviours.

Drips Network: That's the point, this is a subsidized flow where we provide funds for the users. When the funds run out, the users will need to start providing LINK themselves, via `transferAndCall`. It is a good point, acknowledged.

Spearbit: Acknowledged.

5.3.4 Updating the AnyApi operator info for the `RepoDriver` takes at least `2` days on main net

Severity: Informational

Context: RepoDriver.sol#L197-L203

Description: The current `admin` of the deployed `RepoDriver` on the mainnet is a `Timelock` contract with the delay of `2 days`. So incase an update is required, it would take at least `2 days`.

Also the `admin` of the `Timelock` contract is the `Governor` contract.

Recommendation: In case of a needed update, it would be probably best to pause the driver quickly.

Spearbit: Why is the admin not allowed to update the AnyApi config while the RepoDriver is paused?

Drips Network: That's a good question, it indeed doesn't seem necessary.. The configuration update on the paused contract will be fixed in a future release.

Spearbit: Acknowledged.

5.3.5 Checking against `address(0)` is missing in `onlyOwner(...)`

Severity: Informational

Context: RepoDriver.sol#L111

Description: Checks against `address(0)` is either missing here or in `ownerOf(accountId)`. `_msgSender()` could potentially be `address(0)` if a trusted forwarder is used does not handle meta-tx correctly. Although this is not the case if the trusted forwarder used is `Caller`.

Recommendation: It might be best to add the check or at least document and comment about this potential issue.

Drips Network: Acknowledged, but probably won't fix. This is risk built into ERC-2771 standard, and even OZ implementation accepts it. I don't think that the extra checks are worth the added complexity and gas cost, `Caller` is a fairly simple contract and easy to prove that it indeed includes the correct sender.

Spearbit: Acknowledged.

5.3.6 There is no endpoint to cancel requests for `RepoDriver`

Severity: Informational

Context: RepoDriver.sol

Description: There is no endpoint to cancel requests for `RepoDriver`.

Recommendation: It would be best to implement canceling requests that have not been fulfilled by the node operator to get the escrowed `LINK` token back. Perhaps this could be a privileged endpoint or one can define a mechanism to keep track of what entity made the original request and only that entity would be able to cancel the unfulfilled request.

Drips Network: That's a good idea, but we're planning to migrate away from AnyApi, so we probably won't add this feature.

Spearbit: Acknowledged.

5.3.7 Add more details to and fixed the comments

Severity: Informational

Context: Drips.sol#L125, Drips.sol#L248, Drips.sol#L253, Streams.sol#L692, Splits.sol#L177, Splits.sol#L219, Streams.sol#L230-L233, Streams.sol#L254-L259, Streams.sol#L609, Streams.sol#L987, Streams.sol#L1024,

Description/Recommendation:

- Drips.sol#L125: `splits` is the sum of all the splittable and collectable amounts for all account ids per `ERC20` token.
- Drips.sol#L248: `streams` needs to be `splits`

```
- /// @notice Increases the balance of the given token currently stored in streams.
+ /// @notice Increases the balance of the given token currently stored in splits.
```

- Drips.sol#L253: `streams` needs to be `splits`

```
- /// @param amt The amount to increase the streams balance by.
+ /// @param amt The amount to increase the splits balance by.
```

- Streams.sol#L692: `realBalanceDelta` is the capped delta to make sure the `balanceDelta` used does not cause the current effective balance to go below `0`.
- Splits.sol#L177: `balance.collectable += collectableAmt` can potentially wrap around if the total supply of the `ERC20` token can be more than `type(uint128).max`. There is already a warning for the used tokens to not have more than `type(int128).max` supply. But maybe adding a comment here would be useful too.
- Splits.sol#L219: `weight / _TOTAL_SPLITS_WEIGHT` up to some errors (`-/+ 1 wei`)
- Streams.sol#L230-L233: The name and description of this field `nextSqueezed` are not accurate.

It might be best to instead have something line:

```
/// @notice The next squeeze start cap lower-bound.
/// Each `N`th element of the array is the squeeze start cap timestamp lower-bound
/// of the accountId's (which is the stream sender) `N`th streams configuration (StreamHistory)
// in effect in the current cycle that the `_squeezeStreamsResult(...)` will be called.
mapping(uint256 accountId => uint32[2 ** 32]) squeezeStartCapLowerbound;
```

Note for the squeezed configurations the corresponding array elements are set to the timestamp the squeezing was performed. This will make sure we don't double-squeeze for the same time interval.

Also note `N` in the comments is not measured against the total number of stream configurations in the whole history. The max value for it is always relative to the number of configurations for the current cycle.

- Streams.sol#L254-L259: The naming and the comments are not detailed enough. These deltas actually relate more to rates of streams per cycle (rate might be a better word).

`this` and `next` prefixes are also a bit misleading when one actually finds out how they are used. The `next` part actually takes care of the start and end of the streams where one needs to adjust the amount partially. And `this + next` is the rate of a stream per cycle. More details in "Stream Deltas Appendix".

- Streams.sol#L609: change `timestamps` to `timestamp` (remove the `s` at the end):

```
- /// @param timestamp The timestamps for which balance should be calculated.
+ /// @param timestamp The timestamp for which balance should be calculated.
```

- Streams.sol#L987: mention that the changes are only to the deltas and the `nextReceivableCycle` for the current and the new receiver states.
- Streams.sol#L1024: consider fixing the following comment:

```
- // Limit picking both curr and new to situations when they differ only by time
+ // Limit picking both curr and new to situations when they differ only by streamId, start and/or
↪  duration
```

Drips Network:

- Regarding RepoDriver.sol#L217: AnyApi doesn't support node operators setting the fees in the calling contracts. The contract is supposed to know upfront whether the fee is enough, and the operator may or may not accept a request with the attached fee.
- Regarding Streams.sol#L230-L233: I'm not convinced that this is clearer, I don't understand what "start cap lower bound" means even though I wrote this code
- Regarding RepoDriver.sol#L253-L254: AnyApi doesn't support error handling. If a request can't be fulfilled, the node operator just doesn't do anything. It's documented in `onTokenTransfer`, which is the only entry point for the users that can be used to pay for their own ownership updates:

```
/// The received tokens are never refunded, so make sure that
/// the amount isn't too low to cover the fee, isn't too high and wasteful,
/// and the repository's content is valid so its ownership can be verified.
```

The rest will be fixed in the next release.

Spearbit: Acknowledged.

5.3.8 The value for `MAX_TOTAL_BALANCE` is manually set

Severity: Informational

Context: Drips.sol#L74-L75

Description: The value for `MAX_TOTAL_BALANCE` is manually set. This value is supposed to be the minimum of `_MAX_STREAMS_BALANCE` and `_MAX_SPLITS_BALANCE` with the current implementation the minimum is `_MAX_STREAMS_BALANCE` and so this value is correctly set.

Recommendation: It would be best to set this value dynamically or at least have unit tests to make sure the correct value is set.

Drips Network: It should be something like `_MAX_STREAMS_BALANCE < _MAX_SPLITS_BALANCE ? _MAX_STREAMS_BALANCE : _MAX_SPLITS_BALANCE`, but Slither breaks on it and I haven't found a workaround. Probably an immutable explicitly initialized in constructor could work here?

A test will be added in a future release, and a ternary operator initialization when slither fixes the issue.

Spearbit: Acknowledged. Having a unit test is recommended.

5.3.9 Rename `sv` in `callSigned`

Severity: Informational

Context: Caller.sol#L209

Description/Recommendation: Consider renaming `bytes32 vs` into `bytes32 sv` to indicate the way `v` and `s` are encoded in a `bytes32` with `v` being the highest bit. This naming would also be consistent with other libraries such as OpenZeppelin.

Drips Network: Will be fixed in a future release.

Spearbit: Acknowledged.

5.3.10 Inconsistent implementation of `emitAccountMetadata`

Severity: Informational

Context: AddressDriver.sol#L170-L172, ImmutableSplitsDriver.sol#L83, NFTDriver.sol#L361-L366, RepoDriver.sol#L528-L535

Description: `RepoDriver`, `NFTDriver`, and `ImmutableSplitsDriver` only call `Drips` to emit the account metadata if the provided array is non-empty. The non-emptiness check of the array is missing in `AddressDriver`.

Recommendation: For consistency only, the following could be implemented for `AddressDriver` as well:

```
if (accountMetadata.length == 0) return;
drips.emitAccountMetadata(_callerAccountId(), accountMetadata);
```

The execution of `forge snapshot --diff` yields:

```
testSetSplits() (gas: 1 (0.002%))
testEmitAccountMetadata() (gas: 20 (0.067%))
Overall gas change: 21 (0.000%)
```

The higher gas price might be due to contract size. If unit tests for empty `accountMetadata` are missing they should be added.

Drips Network: Will be fixed in the next release.

Spearbit: Acknowledged.

5.3.11 Use the internal function/hook `_drips()`

Severity: Informational

Context: AddressDriver.sol#L163, AddressDriver.sol#L171, ImmutableSplitsDriver.sol#L81-L83, NFTDriver.sol#L219-L220, NFTDriver.sol#L337, NFTDriver.sol#L365, RepoDriver.sol#L405-L406, RepoDriver.sol#L519, RepoDriver.sol#L534

Description/Recommendation: It might be best to use the internal hook `_drips()` for consistency and future refactoring. There is no change in gas costs according to `forge snapshot --diff`. This will make sure in case in the future there needs to be extra logic associated to calling `Drips` it will already be refactored into this internal function.

```
function _drips() internal view override returns (Drips) {
    return drips;
}
```

Drips Network: I would prefer to use `drips` than `_drips()`, because it's cleaner and easier to read. These are fairly simple contracts, so refactoring them to have a more complex `drips` address retrieval procedure won't be a problem. `_drips` only exists to avoid passing the address into `DriverTransferUtils`, it's not supposed to be used outside of this scope.

I think that dropping `_drips()` in favor of just passing the `Drips` address into the helpers functions would be a better approach. Will be applied in the next release.

Spearbit: Acknowledged.

5.3.12 `currCycleConfigs` state variable name is inaccurate and can be confusing

Severity: Informational

Context: Streams.sol#L244-L245

Description: The name `currCycleConfigs` and the comment above don't accurately describe the content of the state variable. Current indicates relative to the current time or cycle. The state variable represents the number of configs at the `updateTimetimestamp`, which can be in a previous cycle. Therefore `lastUpdatedCycleConfigs` would be more accurate.

Recommendation: Consider renaming the state variable to `lastUpdatedCycleConfigs`

Drips Network: Will be fixed in the next release.

Spearbit: Acknowledged.

5.3.13 Double delegation in `onlyHolder()` modifier.

Severity: Informational

Context: NFTDriver.sol#L56

Description: The `onlyHolder()` modifier does not only allow the owner of the NFT but also any approved operator of the `msgSender()`. This allows for possibly unintended double delegation.

For instance the owner of the NFT could set an approval for address A. This address A could have set address B as authorized in the Caller contract. The result is that address B is able to control the account, whereas the owner merely set an approval for address A.

Recommendation: Consider changing the name of the modifier to indicate approvals and their Caller delegates are included if this is wanted or remove the double delegation by only allowing the owner of the NFT and any address authorized in the Caller contract.

Drips Network: Will rename to `onlyApprovedOrOwner` in the next release. The `Caller` double delegation is there by design, it allows e.g. batching or scripting operations on NFT-based identities that aren't owned, but approved.

Spearbit: Acknowledged.

5.3.14 Risk of change to critical constant values during upgrade.

Severity: Informational

Context: Splits.sol#L25, Streams.sol#L156, Streams.sol#L163

Description: `_TOTAL_SPLITS_WEIGHT` is set as a constant in the implementation contract. During an upgrade if this value would be changed any pre-existing split has their individual weights set relative to the old value and any calculation with the new value would result in incorrect amounts. Care should be taken this value is not modified during an upgrade.

Other constants with possible similar impact would be `_AMT_PER_SEC_MULTIPLIER` and `_cycleSecs`.

Recommendation: Implement a check on important constant and immutable values during upgrade or include a note in the code.

Drips Network: Acknowledged.

Spearbit: Acknowledged.

5.3.15 `Drips.withdraw`: concerns about potential future flaws

Severity: Informational

Context: Drips.sol#L305

Description: `Drips.withdraw` is callable by anyone. This means that anyone can pull the withdrawable balance of tokens from the `Drips` contract, which is defined as `uint256 withdrawable = _tokenBalance(erc20) - streamsBalance - splitsBalance`.

It is important to mention that during this engagement we were not able to find any vulnerabilities around this function.

This design means that every code flow that decreases `streamsBalance` or `splitsBalance` will have to also include a call to `withdraw` right after that. Moreover, there should not be a way to reenter the `Drips.withdraw` function, or at least not before the original caller had received their tokens from the `withdraw` function.

Recommendation: To mitigate any potential risks around this function and any future code flows that includes this function, consider implementing the following:

1. Document this potential vulnerability in the internal documentation of the code.
2. Add fuzzing invariants to the CI/CD to make sure that introducing attack vectors is impossible in new versions of the code. This can be achieved by using stateful fuzzing to test that no state can be reached where calling `Drips.withdraw` directly (with no driver in between) will withdraw funds to an external user.
3. Pay attention to these potential future vulnerabilities in internal code reviews.
4. Consider Adding a reentrancy guard protection to `Drips.withdraw`. Although it is not necessarily needed in the current version of the code and will only solve reentrancy issues, it can decrease the potential surface area for attacks.

Drips Network: A large obstacle for any attack where in a single transaction a 3rd party withdraws between funds, become withdrawable in Drips and they are actually withdrawn, is that on withdrawal the caller needs to specify exactly how much should be withdrawn, and if it's impossible, an exception is thrown. This still has a potential to block funds forever, but at least they can't be stolen that way.

This potential vulnerability is already documented, see `Drips.withdraw`, `Drips.collect`, `Drips.give` and `Drips.setStreams`. Each of these functions describes what's going on and what's the risk around `withdraw`.

Currently, the addition of fuzzing invariants is being worked on.

It is worth considering adding a reentrancy guard protection if we change the `Drips.withdraw` logic, I agree that as of now it would be a waste of gas.

We acknowledged to pay attention to these potential future vulnerabilities in internal code reviews.

Spearbit: Acknowledged.

5.3.16 Avoid floating compiler versions

Severity: Informational

Context: Global scope

Description: Avoid using floating pragmas for non-library contracts. While floating pragmas make sense for libraries, as they allow them to be included with multiple versions of applications, they may pose a security risk for application implementations. A vulnerable compiler version might be inadvertently selected, or security tools might fall back to an older compiler version, checking a different EVM compilation that is ultimately deployed on the blockchain.

Drips Network: The way contracts are organized now requires some degree of freedom of choice of the compiler version. E.g. in order to call `Drips` without manual API copying, a 3rd party needs to import `Drips.sol`, and it must be able to work with the 3rd party's compiler. If for each contract we had an interface defined in a separate file, these interface files could indeed be treated as "libraries" with floating compiler versions while the implementations could have a fixed version.

While the versions are floating, their minimum is the version we're planning to use. It's impossible to use an older version, the freedom is only about using newer, potentially safer versions. Is allowing compilation with future versions of Solidity a bad thing? The only problem I see is if a 3rd party decides to use an old version of solc in their new project and tries to use `Drips`, but why would they do that?

Spearbit: The floating pragma used is `^0.8.20`. The foundry config used is:

```
[profile.default]
solc_version = '0.8.20'
evm_version = 'Shanghai'
...
```

`Shanghai` introduces EIP-3855 for `PUSH0` and also `solc` starting `0.8.20` uses `PUSH0` if `evm-version` is `Shanghai` or more recent. And so the contracts cannot be compiled with this current configurations for the other chains where `PUSH0` is not implemented.

Drips: That's true about Shanghai not being supported on all chains. How would you tackle this problem?

Spearbit: Either fixed the pragma version to `0.8.19` or provide documentation for the devs in the `README.md` and other places that in case they would want to deploy the contracts on other chains they would need to do their due diligence of finding out whether the chain in the question supports `PUSH0` or not. If not they would need use `evm-version` set to `paris` or the forks before then.

Drips: If I fix pragma to `=0.8.19`, I won't be able to compile code with solc 0.8.20 and Shanghai target to support `PUSH0` on Ethereum, so it's an unacceptable approach. Solc 0.8.20+ does support compilation for earlier EVM versions, so it can be used to target both chains with and without `PUSH0`. E.g. `forge create` has the `--evm-version` parameter that can be used to override `foundry.toml` and deploy on pre-Shanghai chains.

Spearbit: Yes, it would also depend on your requirements on whether you would like to take advantage of `PUSH0` opcode on the chains that would support it.

Drips: Of course I want to use `PUSH0` wherever it's possible, it would be a huge gas optimization opportunity miss not to do that. Our official Ethereum mainnet deployment was made with `PUSH0` enabled, it reduced gas usage, reduced the bytecode size and allowed increasing compiler runs.

Spearbit: Acknowledged.

5.4 Appendix

5.4.1 Stream Deltas

Context: Streams.sol

Description:

```
struct AmtDelta {
    int128 thisCycle;
    int128 nextCycle;
}

struct StreamsState {
    ...
    mapping(uint32 cycle => AmtDelta) amtDeltas;
}
```

parameter	description
`erc20`	a fixed `ERC20` token
`rid`	a fixed receiver account id
ΔA	`_streamsStorage().states[erc20][rid].amtDeltas`

parameter	description
a_i	`_streamsStorage().states[erc20][rid].amtDeltas[i].thisCycle`
b_i	`_streamsStorage().states[erc20][rid].amtDeltas[i].nextCycle`
R_i	$a_i + b_i$ the total rate of the cycle i
R_W	$\sum_{c \in W}(a_c + b_c)$ here W is a set, and so R_W is the total rate of flow for that set
t_e	capped end timestamp for a stream (depends on the context)
t_s	capped start timestamp for a stream (depends on the context)
r	the amount per second for a stream
S	`_cycleSecs`
F	`fromCycle` (depends on the context)
T	`toCycle` (depends on the context)
t_{now}	current timestamp

- `_setStreams(...)`

When a new stream gets added to the deltas for a receiver `rid` the a and b transition like below:

$$(a_i, b_i) \to (a_i, b_i) + \left(+\left\lfloor \frac{Sr}{10^9} \right\rfloor - \left\lfloor \frac{(t_s \,\%\, S)\, r}{10^9} \right\rfloor, +\left\lfloor \frac{(t_s \,\%\, S)\, r}{10^9} \right\rfloor \right)$$

$$(a_j, b_j) \to (a_j, b_j) + \left(-\left\lfloor \frac{Sr}{10^9} \right\rfloor + \left\lfloor \frac{(t_e \,\%\, S)\, r}{10^9} \right\rfloor, -\left\lfloor \frac{(t_e \,\%\, S)\, r}{10^9} \right\rfloor \right)$$

Above the values for i and j are the corresponding cycles for t_s and t_e ($c = \lfloor \frac{t}{S} \rfloor + 1$). As one can see the overall effect of the above transition to the sum of all deltas over the whole cycles is `0`:

$$\Delta \left[\sum_c (a_c + b_c) \right] = 0$$

The same is true when streams are removed the effect on the sum of deltas is `0`. The same is true when one receives or squeezes a stream as we will see below. And so we always have:

$$R_{\mathbb{Z}} = \sum_c (a_c + b_c) = 0$$

- `_receiveStreams(...)`

$$(a_T, b_T) \to \left(a_T + \sum_{c \in [F,T)} (a_c + b_c),\, b_T \right) = \left(a_T + R_{[F,T)}, b_T \right)$$

$$\forall c \in [F, T) : (a_c, b_c) \to (0, 0)$$

The $R_{[F,T)}$ have been added to a_T as for all the streams that have not been ended their rates would still need to be accounted for.

and so:

$$\Delta\left[\sum_c (a_c + b_c)\right] = 0$$

Let's look at `_receiveStreamsResult(...)`:

```
function _receiveStreamsResult(uint256 accountId, IERC20 erc20, uint32 maxCycles) ...
{
    unchecked {
        ...
        for (uint32 cycle = fromCycle; cycle < toCycle; cycle++) {
            AmtDelta memory amtDelta = amtDeltas[cycle];
            amtPerCycle += amtDelta.thisCycle;
            receivedAmt += uint128(amtPerCycle);
            amtPerCycle += amtDelta.nextCycle;
        }
    }
}
```

One can show that:

$$\texttt{amtPerCycle} = R_{[F,T)}$$

and

$$\texttt{receivedAmt} = \sum_{c\in[F,T)} \Big((T-c)(a_c+b_c) - b_c\Big) = \sum_{c\in[F,T)} \Big((T-c)R_c - b_c\Big)$$

Above makes sense as for each cycle c we would want to add R_c to the received amount depending on how far behind that cycle is from the `toCycle` T and that is why there is the factor $T-c$. The subtraction for b_c comes from the fact that we would need to take care of the stream start and end amounts where one needs to account for $\left\lfloor \frac{(t \,\%\, S)r}{10^9} \right\rfloor$ components (which can have positive or negative sign).

and R_c is the sum of all the rates corresponding to cycle c:

$$R_c = \sum \epsilon \left\lfloor \frac{Sr}{10^9} \right\rfloor$$

where the ϵ factor can be either 1 or -1 depending on the stream with the rate r.

- `_squeezeStreams(...)`

In `_squeezeStreams(...)` the deltas are updated as follows:

```
uint32 cycleStart = _currCycleStart();
_addDeltaRange(
    state, cycleStart, cycleStart + 1, -int160(amt * _AMT_PER_SEC_MULTIPLIER)
);
```

Here if $t_{now} = (c-1)S + k$, then `cycleStart` would be $(c-1)S$ and both `cycleStart` and `cycleStart + 1` would end up on the same cycle c (this is due to the fact that it is required that $S > 1$). And so the over all net effect on the deltas would (only the delta related to c is changed):

$$(a_c, b_c) \to (a_c, b_c) + B(-1, 1)$$

where B is the total squeezed `amt`. Note that the net of the flow rate $-B + B$ is zero so this would not contribute to the next receivable cycles, but the second component B would make sure that this amount is subtracted from the next receivable amounts in b_c.

And again one can see that:

$$\Delta \left[\sum_{c} (a_c + b_c) \right] = 0$$

5.4.2 Fuzzing report

Context: Global scope

Description: The fuzzing report of the codebase is shown below:

Property	Result
Withdrawing directly from Drips should always fail	PASSED
`amtPerSec` should never be lower than `drips.minAmtPerSec()`	PASSED
Total of internal balances should match token balance of Drips	PASSED
The sum of all `amtDelta`s for an account should be zero	PASSED
Giving an amount <= token balance should never revert	PASSED
Test internal accounting after squeezing	PASSED
`drips.squeezeStreamsResult` should match actual squeezed amount	PASSED
Squeezing should never revert	PASSED
Test internal accounting after receiving streams	PASSED
If there is a receivable amount, there should be at least one receivable cycle	PASSED
`drips.receiveStreamsResult` should match actual received amount	PASSED
Receiving streams should never revert	PASSED
Test internal accounting after splitting	PASSED
Splitting should never revert	PASSED
Test internal accounting after collecting	PASSED
Collecting should never revert	PASSED
Setting streams with sane defaults should not revert	PASSED
Adding streams with sane defaults should not revert	PASSED
Removing streams should not revert	PASSED
Test internal accounting after updating stream balance	PASSED
Updating stream balance with sane defaults should not revert	PASSED
Withdrawing all stream balance should not revert	PASSED