zkGit

Marcin Górny
marcin@hungrycats.studio
Hungry Cats Studio

Antonio Mejías Gil
anmegi.95@gmail.com
Hungry Cats Studio

Hossein Moghaddas
autquis@gmail.com
Hungry Cats Studio

November 27, 2023

Abstract

This document explores the problem of succinctly and anonymously proving Git contributions using succinct non-interactive arguments of knowledge (SNARKs). After a brief introduction to the technicalities of Git objects and their signatures, we propose several solutions revolving around the concept of incrementally verifiable computation (IVC) and discuss their security and applicability. Finally, some limitations imposed by real-world usage of Git, together with some potential workarounds.

Keywords: Git, contribution, signatures, proof of knowledge, SNARK, IVC, Nova

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 3
- 1.2 Project Motivation and Examples . . . 3
- 1.3 Deploying zkGit on Ethereum . . . 4
- 1.4 Related works . . . 8

2 Preliminaries **9**
- 2.1 Commit objects in Git . . . 9
- 2.2 Practical example: replicating GPG Git signatures . . . 11

3 System Definition **12**

4 Incrementally Verifiable Computation for zkGit **14**
- 4.1 The Choice of an IVC Scheme . . . 15
- 4.2 Proof of Knowledge through Re-signing . . . 16
  - 4.2.1 Security Considerations . . . 19
- 4.3 Proof of Knowledge through Key-pair Relation . . . 23

5 Merge policies **24**

6 Conclusions **29**

References **31**

1 Introduction

1.1 Problem Statement

The Git protocol is the most popular version control system used by developers worldwide [Sta]. Whether self-hosting or using a centralized provider like GitHub or Bitbucket, teams can collaborate on writing code efficiently and often sharing their repositories publicly for others to read, use, and modify.

Open-source development is an important part of software engineering, providing public auditability, ease of integration, and speed of development. Open-source repositories rely on one or more maintainers (typically the repository originators or long-time contributors) to push the development forward, propose directions as well as curate the incoming community contributions. The role of a repository maintainer or a frequent contributor brings about a certain status to the developers, especially for well-known projects or important/complicated changes required to implement a new feature, fix a bug, etc.

Hosting platforms such as GitHub enhance this effect by allowing users to display their activity on their public profile, and even take a step towards gamifying the experience by issuing badges[1] (e.g., Pull Shark). By interacting with repositories within a specific subject of interest, users also learn by experience the handles of other notable users, via reading Issues, reviewing Pull Requests, or seeing their changes via `git blame`, often integrated into code editors. Aside from an online reputation, a strong Git contribution history can carry real-life benefits, such as competitive advantage when applying for jobs or public funding. It can also translate into an advantage when participating in discussion forums.

One important issue with the status quo of contributing to open-source projects is the privacy vs. reputation tradeoff. Some developers may wish to remain anonymous and contribute to each repository with a separate account (or even more extremely, separate account per contribution) in order to break the link between their various profiles and thus obscure their identity. Note that contribution metadata, such as e.g., IP address or stylometry, are additional factors that a user needs to take into account in order to remain truly anonymous against all actors. By definition, such users lose out on the public reputation benefits of maintaining all their activity in one public profile, visible to all. With zkGit, we wish to bridge this gap and provide a novel method to interact with Git anonymously, while providing a publicly verifiable and trustless reputation score.

1.2 Project Motivation and Examples

We propose zkGit, an end-user app letting the users prove their valuable contributions to popular Git repositories, without revealing their identity. The potential applications of zkGit are numerous: from proving open-source contributions to your potential employer, to integration with bug bounty programs or grants platforms.

Privacy Use Case. For clarity, let us consider a privacy-preserving example. Imagine that Alice was using her `@anonymous_alice` GitHub account to contribute to the Tornado Cash repositories and made 20 commits over 2 years. She is now in need of a new job and is worried that revealing

[1] https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-profile/customizing-your-profile/personalizing-your-profile#earning-achievements

that she is `@anonymous_alice` could endanger her freedom and put her at risk of imprisonment. Instead, when she applies for a new job, she could use zkGit to prove that she:

- made > 3 PRs to Tornado Cash repository,
- or is in Top-5 contributors to one of the private payment repositories: {Tornado Cash, Zcash, Monero},
- or added or modified > 200 lines of Solidity code in some repository with > 100 GitHub stars,
- or co-authored > 3 PRs labelled "priority: top; difficulty: hard" in some repo that Tornado Cash has a dependency on;

depending how much she is willing to reveal.

Funding Use Case. A second example of how zkGit would be used is within a Grants platform. Consider https://github.com/rust-rse/reed-solomon-erasure repository, which is at the time of writing looking for new maintainers. Picture Bob, an aspiring Rust developer that wants to contribute to the ecosystem and take on the role of the repository maintainer. He could use a Grants platform integrated with zkGit to create a funding request. The platform, via its verifier smart contract deployed on Ethereum, would automatically verify his contributions and pay him for the hard work done. The request could be conditioned on:

- Opening and having merged > 3 PRs to https://github.com/rust-rse/reed-solomon-erasure repository within 6 months,
- Opening 2 issues that are labelled as "Critical" by any of the other repository authors,
- Reviewing (co-authoring) > 1 PR / month.

If he fulfils some or any of the above condition, he could make a succinct proof and submit it on-chain to verifier contract. He does not even need to make all of the 3 PRs with the same GitHub handle! (Note that in this example, zero-knowledge is optional—the key here is succinctness)

1.3 Deploying zkGit on Ethereum

To give shape to the protocol described above we should consider the practicality of deploying it as an end-user application. This entails discussing the various limitations of the deployment environment, a blockchain, and how it affects the design choices of zkGit. We will focus our discussion on Ethereum as the most widely used smart-contract chain, but in principle, the argument broadly applies to other chains.

We split this section into an overview of the protocol, followed by the description of the types of interactions that zkGit parties engage in. We then conclude the section with the discussion of three important design considerations for our protocol: communication/storage costs, computation costs, and trust assumptions.

10,000-feet Overview. On a high level, the instantiation of zkGit protocol consists of a (series of) smart contract(s) initialized with a list of Git repositories and some information about each repository (metadata). Both the repository list as well as the metadata can, and should, be updated periodically for the protocol to offer the best user experience. The on-chain state also includes a public ledger of contribution (a.k.a. reputation) scores, initialized as empty.

The fundamental logic of the smart contract is the verification of zkSNARK proofs attesting to the statements of the form: "Given a git repository X, I have authored N commits in that repository." Any Ethereum user is allowed to interact with the zkGit protocol by submitting such a (valid) SNARK proof.

Upon the successful verification of a proof for a given statement (both supplied by the user) against the verifier's stored information about the repository in question, the smart contract updates the reputation score associated with the user. Other smart contracts can now read the zkGit storage and access the contribution scores of users of interest, potentially adding an extra layer of complexity to rate the contributions based on some protocol-specific criteria (e.g., contract A gives 2x weight to repositories within the `https://github.com/ethereum` organization).

Types of interaction. Based on the above overview, the following summarizes the different types of interactions with the zkGit protocol:

- `repo_add`: **Add a new repository**. While the protocol might be initialized with a comprehensive list of repositories of interest, new repositories will inevitably be created; or other existing repositories might become relevant to new zkGit users—in any scenario, the protocol should allow for updating the list of repositories it tracks.

- `metadata_update`: **Update the repository metadata**. After adding a new repository (either at protocol genesis or per the step above) to the tracking list, any active repository will have its history updated with new commits. zkGit smart contract (and any on-chain contract) is not able to learn about these metadata changes unless such an update is explicitly invoked by an external call to the contract. Allowing on-chain access to off-chain data (such as git repositories) is referred to as the job of an oracle. Note that while in practice the API might allow for a combined operation of `repo_add & metadata_update`, for the ease of exposition we explicitly separate the two here: `repo_add` does not carry any metadata.

- `submit_proof`: **Submit a proof**. The smart contract should inherit 2 key properties of the underlying zkSNARK verifier. We state these here, contextualized for clarity:

  - Completeness: a valid proof of git contribution should always be accepted by the smart contract.
  - Soundness: an invalid (or faked) proof should be rejected (due to the probabilistic nature of SNARKs, we require the invalid proofs to be rejected with high probability)

- `query_score`: **Querying the contribution scores**. Any actor, whether another on-chain contract or an off-chain client, should be allowed to read and query the contribution scores stored as part of the zkGit state. The results is stored as key-value pairs, with the key "(user, repository)" and value "latest score".

Communication and storage. Due to the potentially high transaction fees and costs associated with the on-chain data storage, one of the objectives of zkGit is to minimize the amount of communication (amount of data transmitted in each transaction) associated with `metadata_update` and `submit_proof`.

Clearly, for the `submit_proof` action, the first line of optimization is the size of the submitted proof itself—fortunately, SNARKs are designed to achieve just that, with proof sizes sublinear in the size of the computation and as small as constant size for some protocols. However, we also need to take into account the fact that the prover submits, along with the proof itself, the statement that he is trying to convince the SNARK verifier of. In the case of zkGit, as described in the overview of this section, the statement must contain, at a bare minimum, a reference to the repository (e.g., some sort of UUID that is much smaller than, for example, the repository URL), as well as the claimed contribution count n. The verifier should also have some mechanism to tie the submitted statements and proofs to some public reference, lest the prover is able to convince the verifier of any unrelated repository **not** tracked by zkGit.

The way we create this link is by leveraging the repository metadata. In the naive approach, the metadata would contain the entire git commit history—basically the output of `git log` stored on-chain. This would ensure correctness, but clearly go against the principle of reducing the required storage (as well as communication). Instead, we propose this metadata to be the digest of the repository's current canonical `HEAD` commit. A `metadata_update` to any of the tracked repositories simply contains the new digest, and can be triggered (externally) as frequently (e.g. with each new commit) or infrequently (e.g. 1x/month) as desired by the protocol users, directly controlling the communication costs. Importantly, we note that the amount of storage **per repository** is constant, as any new update will overwrite the previously stored value. We later propose batching techniques to minimize the **total amount of storage** for the protocol to a single hash digest by using succinct commitments with local openings (e.g., Merkle trees), at a cost of foregoing flexibility.

Computation. Another optimization goal for the protocol is to minimize the computation required to both produce and verify the proofs. In the simplest instantiation of the protocol, the end-user assumes the costs of both of these. Let us break the computation cost in to the following.

- **Proof generation** Clearly to convince the on-chain verifier smart contract of the validity of git contributions, the end-user has to calculate the proof somehow. We explain that, unless we resort to complicated and inefficient MPC techniques, this necessarily needs to happen **locally** for the user. Note that, locally does not necessarily mean on the same device that is used to submit the proof on-chain. More precisely, we require that the end user has full control of, and trust in the device they use for proof generation and as such is not outsourcing proving to a third party. The fundamental reason behind this is that proof of authorship of git commits relies on proof-of-knowledge of the private key that was used for commit signing—which, by definition, implies that the private key is the witness to some argument system (e.g., SNARK) and must be supplied "in plain".

  The above described requirement, quite standard when we care about the zero-knowledge of the prover's inputs, leads to a practical consideration that is worth reiterating here: the resources required for generating a proof should allow for calculating it on consumer hardware.

- **Proof verification** Aside from the hopefully negligible compute costs to produce the proof, the majority of the cost burden on the user will come from paying enough Ether to cover the gas

costs associated with the proof verification.

One of the most widely deployed on-chain verifiers is for the Groth16 proof system, where a single verification call costs 222k gas at a minimum [Ebe21, Section 11.2.1], or $11 on mainnet Ethereum as per the time of writing.

While intuitively the SNARK verification should be **cheaper** than proving, an on-chain verification results in all the nodes in the network re-executing the same state update, hence the cost. We would like to do better, or at least not too much worse that this.

- **Universal circuit** Finally, while this is not strictly a requirement on neither the proving nor verification efficiency, we would like our protocol to NOT have to have a different verifier contract for every statement that we wish to prove.

  For example, we'd like to avoid the situation where for every n (claimed number of commits authored) we have a separate contract. One could of course devise ranges or wrapper contracts that call the original multiple times, but this is impractical and quickly gets out of hand.

  Ideally, the single deployed smart contract would handle all statements that the users wish to prove.

Trust assumption. As briefly mentioned earlier, the Ethereum chain does not have intrinsic knowledge of what is happening in the "web2 world", including GitHub (or other git hosts) repositories. In order to give such extra power to a smart contract, we need to inject the necessary data from the outside by the means of an oracle. Oracles are systems that can provide external data sources to Ethereum smart contracts [AW18, Chapter 11]. In [AW18], we further read:

> If we assume that the source of data being queried by a DApp is both authoritative and trustworthy (a not insignificant assumption), [...] how are we able trust this mechanism?

We can broadly categorize oracles by their underlying mechanism for ensuring data authenticity:

- authenticity proofs
- specialized hardware
- game-theoretic incentives
- a combination of the above

For more details on the mechanisms, we refer the reader to Mastering Ethereum [AW18]. Further down in Section 1.4, we mention a few projects utilizing some of the above approaches.

No technique for bringing off-chain data to smart contracts is perfect. Whether the data needed for proper functioning of zkGit is supplied by an oracle based on Trusted Execution Environment (TEE) or a decentralized network of data providers, we would like to minimize the amount of data that our protocol needs to trust. Intuitively, if we were to rely on a single choice of an oracle for our data, then no matter if we were only relying on 32 bytes of kilobytes of data, our protocol would be equally affected.

Apart from the smaller overhead in communication and storage as outlined above, a lower **amount** of data coming from an oracle allows for certain protection mechanisms to be implemented (or implemented more efficiently). For example, consider the two extreme scenarios, where the oracle is TEE-based:

1. Oracle only supplies the hash digest of the last commit
2. Oracle stores the entire git history on-chain (or even, circumventing the zkGit protocol, users post authenticity proofs of API queries to the git hosting platform directly).

If the oracle were to be compromised, the first scenario offers the succinctness advantage: it is much easier to check the correctness of the oracle's output! An external party can simply consult the git protocol for themselves with one off-chain query. The protection mechanism added here, while itself not perfect, could be a dispute pathway; or an integration of TEE-based oracles into a decentralized and incentivized network. In either case, the on-chain logic for dispute handling or a data aggregation smart contract would be much cheaper in the first scenario.

Lastly, the protocol could deploy an additional, albeit weak, protection against malicious metadata updates: require a cryptographic proof that the new data point is a descendant of the currently stored value. Again, this would be much simpler in the first scenario above, in which case the proof would demonstrate that the current value is part of the hash-chain resulting in the new hash. The weakness of this mechanism is that while it protects against an update with completely unrelated history, it still allows for forked repositories to be considered valid. As a consequence, for repositories that allow force-push to branch `main`, this mechanism would be a dealbreaker and not applicable at all.

1.4 Related works

The Town Crier [ZCCJS16; Tow] protocol is an authenticated oracle for smart contracts, providing a high-trust link between HTTPS-enabled data websites and the Ethereum blockchain, utilizing Intel SGX technology. DECO [ZMMGJ20; Dec] was later introduced, noteworthy for not requiring trusted hardware or server side modifications, and it supports standard TLS websites. Reclaim [Rec] aims to offer publicly verifiable proof of provenance for user-provided data. Clique [Cli] focuses on identity oracles and data retrieval from Web2, employing Intel SGX-based TEE technology. Lastly, TLSNotary [Tls] specializes in cryptographic proofs of authenticity for web data.

All these protocols offer an alternative way to prove facts about Git. A user could use the above services to prove to the Ethereum smart contracts that they've indeed queried the Git host (e.g. GitHub) and received a response certifying to their contributions directly (e.g. `https://github.com/arkworks-rs/algebra/commits?author=mmagician`), rather than the smart contract verifying hash chains and signatures on the individual commits. Assuming that we accept the trust assumptions that the above protocols come with (e.g. TEE, Notary), and which we do not dive into here for brevity, one fundamental difference remains that renders these approaches less Ethereum-friendly: it relies on trusting the API responses of GitHub. Now, the response itself can be (under the protocol's own trust assumption) proven correct - what remains is the single point of failure on GitHub's side. No matter how much guarantees we place in the response itself, the protocol will be vulnerable should the hosting platform decide to return malicious data.

Our approach, and in fact any dApp relying on off-chain data, is at threat of being injected with malicious data from a third party. Our approach aims at reducing the dependency to a bare minimum by only periodic oracle updates. In fact, we suggest the above protocols be used (some of which already are) to augment the oracle network with an additional security layer.

2 Preliminaries

2.1 Commit objects in Git

This section elaborates on the structure of commit objects. Let us start with a short introduction on Git objects.

Git objects. Generally, there are different types of objects in Git such as `commit`, `tree`, `tag`, and `blob`. `tree` and `commit` are the types that we work with in this project. Git uses hashes of these objects as indices for finding them (e.g., same as a hash table). Notice that, in general, this hash function does not have to be cryptographically secure. From now on, the *index* of an object denotes the hash value of that object.

Hashin algorithm. Git uses the SHA1DC algorithm for hashing, also known as Hardened SHA1. Transition to the stronger SHA-256 has been underway for years, but the new standard has not been adopted yet (in particular, by Git platforms). SHA1DC checks whether the received input is susceptible (because of its structure) to a collision attack identified a few years ago [SBKAM17][2]. If it is not, SHA1DC just runs SHA1. If it is, the current default configuration of Git is to compute a "safe hash" which is a modified output that prevents collision [Git]. On uniformly random input, the probability that the output of SHA1DC differs from that of SHA1 is roughly 2^{-63} [SBKAM17].

The following bash script computes the index of a Git object on a GNU/Linux machine.

```
$ sha1sum("$TYPE ${#OBJECT}\0$OBJECT")
```

As for MacOS one can replace `sha1sum` with `shasum`. As explained above, although there is a probability that the output does not match the SHA1DC hash of that string, the probability is negligible.

The structure of commit objects. A commit object is the concatenation of strings corresponding to different elements in a specific order. Most of those strings start with a header (followed by a space), which we specify at the beginning of each point below. Each point corresponds to an `\n`-terminated line unless indicated otherwise:

1. `tree`: Shows the index of the top-level tree object (contains the file structure of the commit).
2. `parent`: Shows the index of the parent commit. If the current commit has no parent (e.g. because it is the first one in its repository), this element does not appear (including header).

   Note that commit merges have more than one parent. In that case, the first listed parent is always the index of the `HEAD` upon which `merge` was called, (i.e. the previous step in the chain hash we are interested in). Observe that, in the output of `git log`, this may not be the immediate predecessor to the merge hash. The subsequent parents are the indices of the `HEAD`s of the branches which were merged onto the current one, in the same order they were passed to the `merge` command.

   For now, the output of the hashing circuit should be fed as the *first* parent of the next execution of the first circuit. However, in the future, this can be changed to allow proving facts about commits in branches which were *merged* onto the branch the verifier knows about, even if their history was lost during the merge.

[2] One can easily test it by replicating the "Attack proof" section in shattered.io

3. `author`: Shows the author and timestamp information with the structure `$name <$email> $timestamp $zone_info`.

4. `committer`: Committer information with the same structure as the author information. The two often coincide (except in the header).

5. `gpgsig`: Shows the signature(s), if any. The header is indeed always `gpgsig`. The signature itself consists of several fields, which we can refer to as a signature wrapper, signature specification, and signature value. All of these fields may vary, with the wrapper being the least likely to do so. Listing 1 demonstrates an example of signatures.[3]

```
----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJXYRjRAAoJEGEJLoW3InGJ3IwIAIY4SA6GxY3BjL60YyvsJPh/
HRCJwH+w7wt3Yc/9/bW2F+gF72kdHOOs2jfv+OZhq0q4OAN6fvVSczISY/82LpS7
DVdMQj2/YcHDT4xrDNBnXnviDO9G7am/9OE77kEbXrp7QPxvhjkicHNwy2rEflAA
zn075rtEERDHr8nRYiDh8eVrefSO7D+bdQ7gv+7GsYMsd2auJWi1dHOSfTr9HIF4
HJhWXT9d2f8W+diRYXGh4X0wYiGg6na/soXc+vdtDYBzIxanRqjg8jCAeo1eOTk1
EdTwhcTZlIOx5pvJ3HO+4hA2jtldVtmPM4OTBOcTrEWBad7XV6YgiyuII73Ve3I=
=jKHM
-----END PGP SIGNATURE-----
```

Listing 1: Example of PGP signatures

6. A blank line

7. Commit message which is a subject and possible a body copy. It may consist of more than one line. The subject is separated from the body copy by a blank line.

From now on, *payload* refers to items 1-5. Listing 2 is an example of a commit object printed by `git cat-file -p COM_ID`

```
tree fe1e7b546d085846365757c889d1239ab8172d69
parent a7f465a09b073581f18a05f0e1d5e8bac98aa9f8
author WorldLeader <worldleader@gmail.com> 1698697449 +0100
committer WorldLeader <worldleader@gmail.com> 1698697449 +0100
gpgsig -----BEGIN PGP SIGNATURE-----
 UmFuZG9tU2lnbmF0dXJlUGxhY2Vob2xkZXIxMjM0NTY3ODkwUmFuZG9tU2lnbmF0
 dXJlUGxhY2Vob2xkZXIxMjM0NTY3ODkwUmFuZG9tU2lnbmF0dXJlUGxhY2Vob2xk
 ZXIxMjM0NTY3ODkwUmFuZG9tU2lnbmF0dXJlU
 -----END PGP SIGNATURE-----

A very important commit
```

Listing 2: Example of commit objects

[3]The apparently blank line in the middle actually consists of a whitespace.

2.2 Practical example: replicating GPG Git signatures

As an example of some of the concepts covered in this section, we provide a code example which uses GPG to sign a (previously signed) commit, recovering the original signature. Looking ahead, this is an important example as it has the essence of Assumption 4.2 required by the approach described in Section 4.2.

Only a terminal with Git and GPG (version 2 or higher) installed is required. The following commands should be executed at the root level of a Git repository, which we assume to contain a signed commit with identifier `COM_ID` (e.g. `HEAD` or the first few characters of its commit index) signed with a GPG key pair the user holds (SSH keys are out of scope for now). Tests have only been conducted with RSA and ECDSA signatures at the time of writing.

1. Extract the commit data to be signed into a separate file:

```
$ git verify-commit --verbose COM_ID 1> commit_data.txt
```

   In the output of the command above, the second line contains the ID(a long hexadecimal string) of the key used to sign the commit, which is necessary for step 3.

2. Obtain the signature timestamp:

```
$ cat commit_data.txt
```

   The timestamp is the unsigned integer after the committer email, for instance: `1699984518`.

3. Re-sign the commit data:

```
$ gpg --detach-sig --armor --default-key KEY_ID --faked-system-time TMS --output \
    new.asc commit_data.txt
```

   Here `KEY_ID` and `TMS` are the key ID and timestamp from previous steps, respectively. The `--faked-system-time` option is crucial, as GPG incorporates into the signature the current system timestamp. This would result in a different signature if we did not mask the current clock.

Optionally, the following steps can be taken in order to verify the re-signing worked:

1. Detach the original signature into a separate file:

```
$ echo '-----BEGIN PGP SIGNATURE-----' > original.asc
  git cat-file -p head | awk \
    '/-----BEGIN PGP SIGNATURE-----/{o=1;next}/-----END PGP SIGNATURE-----/{o=0}o' | \
    cut -c2- >> original.asc
  echo '-----END PGP SIGNATURE-----' >> original.asc
```

2. Check the new signature indeeds with the original one:

```
$ diff original.asc new.asc
```

3. One may also verify new signature is a valid signature on the commit data:

```
$ gpg --verify original.asc commit_data.txt
```

3 System Definition

Recall from Section 1 that the task is for the prover to convince the verifier of their authorship of a specific number of commits in a specific Git repository. To model the prover and verifier, we need to specify:

- the verifier's access to the Git repository,
- the verifier's preprocessing load in the offline phase,
- the amount of data about the repository the verifier stores,
- the degree of anonymity guaranteed for the prover,
- and, assuming the system takes the form of an arithmetic circuit, the structure of the circuit.

This section begins by reviewing the trivial setting for Git verification, referred to as System ∅. It then proceeds to discuss two enhanced systems labeled A and B.

System ∅. In this straight-forward setting, the verifier is not constrained (i.e., not efficient), and no anonymity for the prover is guaranteed. The verifier can independently traverse the entire Git history, scrutinizing the authorship of specific commits. Additionally, they should verify the signatures of the commit objects to ensure their integrity. Note that there is no control over what the verifier may learn beyond the prover's intentions. Also, there is no active prover since the proof (i.e., the Git history) already exists. So, the prover may initiate the verification of authorship for someone else. Observe further that this system resembles an off-chain protocol, requiring the verifier to query the Git protocol for each commit they wish to check authorship of or to store the entire Git history (which is impractical on-chain).

Pros:
- The design is simple.

Cons:
- Either the entire commit history of interest must be stored, or a query to Git must be made for each commit to verify. On chain, the latter requires an oracle network or some trust assumptions.
- Pull-query mechanism, which requires it to ask for some data from the oracle network actively. This is not a blockchain-friendly solution, as the proof verification needs to proceed in steps: proof is posted in block A and the data is requested from the oracle network; in block B the data is sent by the oracles and proof is verified. Crucially, the delay between A and B is out of the verifier's control.

In the following systems, an active prover produces a proof of authorship which is subsequently checked by the verifier. Among them, the authors prefer System A due to its seamless integration with the Git protocol and efficiency of the verifier. Finally, System B, by combining ideas from System ∅ and System A, is a protocol with niche – but potentially useful – use cases.

System A: Index chain. In this design, the verifier is only required to hold a single piece of information: the index ι of the last commit in the Git history of interest. A typical example would be the `HEAD` of the `main` branch of a given repository. The question of how this index can be obtained will be touched upon in a few lines.

A natural approach would be to construct a *monolithic* arithmetic circuit (or R1CS) capturing the notion that the prover is the author of a number of commits in a chain which ends precisely on (a commit with index) ι. Any SNARK for arithmetic circuits could then be used by the prover to convince the verifier of an authorship claim – potentially in zero-knowledge. There are, however, several important obstacles to this design. The most crucial is the fact that, for reasonably sized commit histories, the resulting circuits quickly become unmanageable.

Instead, we propose a modular design with a single circuit which takes some commit data (including the index of a parent commit) and outputs the index of that commit, obtained by SHA1DC-hashing the data, together with an indicator of whether the prover knows the signing key used to sign that commit. The index output by that function can then be fed as the parent index to another instance to the same function. Chaining several such modular circuits can then reproduce the same commit index history of the repository of index. If those circuits are enhanced with a counter keeping track of how many commits the prover knew the signing key for, the verifier only needs to know the final value of this counter and make sure that the final index in the output matches ι.

This modular design perfectly fits the paradigm of incrementally verifiable computation (IVC), where a verifier is convinced about the final result of incremental (i.e. *chained* as above) execution of a given function F. Crucially, recent schemes allow this to happen with minimal work on the verifier's part: essentially, the cost of verifying a *single* correct execution of F and a small amount of overhead. The detailed construction of a function F accomplishing the above goal is the object of Section 4.

To summarize, the prover reproduces the commit history leading to ι by hashing each commit object in it and feeding the resulting index as the parent of the next commit. Furthermore, at each step, they can provide signing data for that commit. Upon verifying the proof, the verifier will learn at how many steps the correct signing data was supplied – but not which steps those are.

It should be noted that, in the basic form of IVC outlined above, the function F must be uniform (i.e. the same) across all steps. As a consequence, the function F representing a single step in the commit history must in fact must contain verification or signature circuits for *all* signing schemes used in the repository. Recently, SuperNova [KS22] introduced a non-uniform IVC scheme, which allows the usage of a different function F at each step while keeping the circuit size constant.

The main limitation of this design is that its security is bound to the hash function: if a dishonest prover can find alternative commit data which they are able to sign and results in the same index as one of the commits in the honest repository history, the verifier will be falsely convinced of the prover's authorship of a commit (or more, if the prover can chain several such custom commits). As explained in Section 2, Git's default hashing function is SHA1DC. While this variant detects collision attacks found in SHA1 [SBKAM17], it is still weaker than, for example, SHA256. Although this does not imply that there is an immediate practical attack on the proposed design, it must be kept in mind. The positive and negative aspects of this design can be summarized as follows:

Pros:
- The verifier needs to store only a single commit index for each repository of interest.
- No preprocessing is required on the verifier's part.

- The identity of the prover can be hidden.
- The prover may choose at which point in the commit history the proof should start: only the end ι is predetermined. This allows for a more tailored workload.

Cons:
- The security is limited to the security of SHA1DC.
- The circuit size *at each step* potentially increases with the total number of signing schemes used in the repository – although non-uniform IVC can mitigate this issue.

For stale/archived repositories, this can be initialized at protocol genesis, but for active repositories we need to periodically update the digest to allow proofs for the most recent history. This responsibility can be outsourced to an oracle network. Note the difference in reliance on the available data: unlike the System $\varnothing$, this approach uses a push mechanism: whenever the oracle network is ready to supply the newest digest, it is posted to the verifier. Any proof can be verified in a single block (note that proofs for history newer than the currently held digest will not be posted by any reasonable client that can, for free, obtain the latest hash from the chain and verify that the proof is for available history).

System B: No index chain. In this system, the public instance to the circuit are the signatures of all the commits, and the verifier queries the Git protocol for acquiring them. For each commit in the range, the circuit contains one signing chip (corresponding the signing algorithm used in the original commit) and a final gate comparing the output of that chip to the public instance. An operator as in the second point System A then returns the circuit output; e.g., total number of valid signatures, or if the number of valid signatures is bigger than some bound.

As opposed to System $\varnothing$, this approach provides a security guarantee for the identity of the prover (i.e., by using a circuit for re-signing), and enables the possibility of limiting the information that the verifier can get; for example, the exact number of commits vs. a bound for them, or giving a range for the location of commits in the history vs. their exact position.

4 Incrementally Verifiable Computation for zkGit

The concept of *incrementally verifiable computation* (IVC) introduced in [Val08] addresses the problem of proving and verifying incremental (i.e. iterated) execution of a function $F\colon S \times \Omega \to S$. It is enough to consider functions F such that the relation $\{\,(s, \omega, F(s,\omega))\,\}$ is NP. Suppose a party has $s_0 \in S$ and $\omega_0, \ldots, \omega_{n-1} \in \Omega$ and wishes to convince another party of the value of

$$s_n = \underbrace{F(\ldots(F}_{n \text{ times}}(s_0, \omega_0), \omega_1), \ldots), \omega_{n-1}) \tag{1}$$

The intermediate values are usually denoted by $s_i = F(s_{i-1}, \omega_{i-1})$ for $0 < i \leq n$, whereas the parties are referred to as the *prover* $\mathcal{P}$ and the *verifier* $\mathcal{V}$, respectively; elements of S, as *states*; and those of Ω, as *witnesses*.

The goal of IVC is to allow the prover to convince the verifier they have witnesses w_i as above resulting in final state s_n from initial state s_0 *without having* $\mathcal{V}$ *perform work that is linear in* n: ideally, verifying three incremental executions of F should be more or less as costly for $\mathcal{V}$ as verifying one hundred (note this is not the case on the prover's side). There are a plethora of applications

for such a scheme. To mention two: asserting execution of verifiable delay functions, and succinct blockchains (where F can model a block transition).

In light of system A from Section 3, it is not difficult to see how the zkGit question lends itself to the IVC approach. Each step of the incremental computation (that is, each execution of F) can represent one commit in the repository $\mathcal{P}$ wants to prove authorship in. The link between each step of the commit chain and the next is precisely the parent commit index field (cf. Section 2.1), which thus constitutes an ideal candidate for the state s. Our aim is therefore to design a function F that captures the notion of whether $\mathcal{P}$ knows the private key used to sign a commit (and they wish to reveal this fact) or not; and to keep track of how many times this was the case across the n iterated executions.

Two designs will be proposed: proof of knowledge through re-signing, and proof of knowledge through key-pair relation. Although the focus is largely put on the former, the advantages and disadvantages of both will be compared, as well as (and indeed in relation to) some preliminary security analysis.

Before that, however, the question of the realisation of IVC must be briefly addressed.

4.1 The Choice of an IVC Scheme

Although a survey of IVC techniques is out of this document's scope, it is worth pointing out that the first attempts at it made use of the concept of SNARK recursion. Essentially, the function F can be *augmented* into a new function $\widetilde{F}$ which, considerably simplifying the matter, performs two tasks: it runs one execution of F (that is, a transition $s_i = F(s_{i-1}, \omega_{i-1})$), and it verifies a proof of a previous correct execution of itself, $\widetilde{F}$. When suitably constructed, this results in the fact that the proof of the *last* execution of $\widetilde{F}$ suffices to convince $\mathcal{V}$ about all incremental executions of F, and hence about the final state s_n. This elegant approach is, however, hardly feasible due to the computational challenges of running the verification for $\widetilde{F}$ algorithm inside a circuit (which is how $\widetilde{F}$ is typically represented), compounded with issues associated to non-native arithmetic. The reader is referred to the introduction of the Nova article [KST22] for more information on these complications.

In that same article, an alternative IVC scheme is proposed which hinges on so-called *folding schemes* (cf. section 3 therein). In essence, a folding scheme for a relation $\mathcal{R}$ consisting of pairs (x, w) allows the reduction of the claim that a prover knows two witnesses w_1, w_2 such that $(x_1, w_1), (x_2, w_2) \in \mathcal{R}$ (with x_1 and x_2 known to both parties) to the claim that they know a witness w' such that $(x', w') \in \mathcal{R}$. The scheme specifies how the prover and verifier must interact to produce the *folded* instance x' and witness w'. Such a scheme is said to be *non-trivial* if the verifier's work to perform the folding and be convinced that the prover has a satisfying witness w' for x' is lower than the the work they would perform to be convinced that the prover has satisfying witnesses w_1 for x_1 and w_2 for x_2. It is furthermore said to be *knowledge sound* if checking the former gives the verifier high confidence in the latter.

In addition to putting forward such a protocol scheme for NP relations, the *Nova folding scheme*, the authors of the cited article construct a general mechanism to compile a folding scheme for NP into an IVC scheme. The resulting *Nova IVC scheme*, which can be rendered non-interactive using standard techniques (the Fiat-Shamir transform), is feasible in practice – as opposed to the recursive SNARKS outlined above. It is fundamentally for this reason that it is our choice for this project.

It should be noted that the only inputs to the general IVC compiler from the article are the choice of a folding scheme (Nova in our case) and the function F itself. In practical terms, this means

that the specifics of incremental computation (including the folding scheme and augmented function $\widetilde{F}$) become transparent to the user of an IVC library, who is simply tasked with providing a suitable representation of F. Several implementations of Nova exist, of which we single out Microsoft's[4].

4.2 Proof of Knowledge through Re-signing

Equipped with a general understanding of IVC, we now present the first of the two designs of F under consideration. The starting point will be the definition of the function itself, followed by an explanation of how it achieves zkGit's aim. Finally, we give an overview the security of the construction. The pros and cons of this design compared to the other alternative are discussed in the next subsection, after the latter has been presented.

As explained at the beginning of this section, we need to define an NP function F representing a step in the commit index chain. Recall (cf. Section 2) that a commit index is the SHA1DC hash value of a commit object, one crucial field of which for our purposes is the signature – which may or may not be present in any given commit. When there is indeed a signature, F should capture whether the witness ω it received contains the private key used to sign the commit in question. The core idea of the "proof of knowledge through re-signing" (PoK-RS) is:

$\mathcal{P}$ can demonstrate they know the signing key by producing the same signature for the same commit data using the honest signing algorithm

The fact that the prover used the *honest* signing algorithm is essential here. It would of course be trivial for a malicious prover to produce any necessary signature if there were not enough restrictions on how to it was actually produced. Fortunately, the IVC scheme convinces the verifier that $\mathcal{P}$ executed F faithfully, and therefore it suffices to incorporate the signing algorithm itself as part of F. Indeed, the only element $\mathcal{V}$ has no knowledge of is the witness ω the prover fed into F (and potentially, for incremental execution, the intermediate states s_i with $i \notin \{ 0, n \}$).

Remark 4.1. Our choice of words so far makes it natural to think of F as a procedural (i.e. step-by-step) function. However, the IVC scheme can only attest to the satisfaction of a ("relaxed", see [KST22, p. 14]) R1CS. To this end[5], one may regard F as an NP relation $\mathcal{R} = \{ (s, \omega, F(s, \omega)) : s \in S, \omega \in \Omega \}$ between input and output, rather than a computation *sensu stricto*. This relation can be expressed as a relaxed R1CS, since the latter is NP-complete.

The distinction between inputs and outputs thus becomes purely semantic, rather than formal. The same IO data can be grouped instead into public (*instance* or, in this case, *states*) and private (*witness*), which does have formal implications for the proving system.

Let $\mathcal{S} = (G, S, V)$ denote a signature scheme, where G, S and V are the key-generation, signing and verification algorithms, respectively. Instead of regarding S as a randomized algorithm $\sigma \leftarrow S(\kappa_s, \mu)$ producing a signature σ given a signing key κ_s and message μ, consider it to be a deterministic algorithm computing $\sigma = S(\kappa_s, \mu, \rho)$, where ρ represents a randomness tape.

Since PoK-RS relies on the prover being able to re-sign commits they made in the past, we introduce under the following assumption on $\mathcal{S}$:

[4] https://github.com/microsoft/Nova.

[5] We are glossing over the fact that the IVC scheme will not directly run F but an augmented variant $\widetilde{F}$, which performs F as well as other computations. However, the issue being brought up still stands.

Assumption 4.2 (Replicability Assumption)**.** There exists an efficient deterministic algorithm $A_S = A_S(\kappa_s, \mu, \sigma)$ such that, for any signing key κ_s, message μ and randomness ρ, one has

$$S(\kappa_s, \mu, A_S(\kappa_s, \mu, S(\kappa_s, \mu, \rho))) = S(\kappa_s, \mu, \rho).$$

This captures the notion that, given a valid signature, along with the signing key and original message, one can recover a randomness tape which yields the same signature (for the same key and message). A slightly stronger (but perhaps clearer) assumption is to require that one can recover *the original* randomness, i.e. to replace the above equation by $A_S(\kappa_s, \mu, S(\kappa_s, \mu, \rho)) = \rho$. Although the weaker version suffices for our purposes, it is likely not much weaker in practice. In other words, it is reasonable to think that if one is able to efficiently recover any randomness that produces the same signature, it should be possible to recover the original randomness.

We are now in a position to define the function F, which depends on the signing scheme and is therefore denoted by $F_{\mathcal{S}}$[6]. Note that it depends on the choice of an A_S as in Assumption 4.2, which we omit in the notation. The input and output of $F_{\mathcal{S}}$ have the following structure (the colon after a symbol introduces that symbol's type):

- **Public input**: state s (first part of the instance)
  - An *input counter* $\tau \in \mathbb{N}$.
  - A *first parent commit index* ι: SHA1DC hash.
- **Private input** ω (witness)
  - A *stripped commit object* γ: string. This amounts to a commit object as in Section 2.1 minus the first parent commit index and the signature. In other words, it contains the secondary parent commit indices (if any), author and committer data, subject and message.
  - A *signature* σ: signature type of $\mathcal{S}$.
  - A *signing key* κ_s: signing key type of $\mathcal{S}$.
  - A *selector bit* $\beta \in \{0, 1\}$.
- **Public output**: state s' (second part of the instance)

  Note that, by the definition of IVC, s' must have the same structure as s.
  - An *output counter* $\tau' \in \mathbb{N}$.
  - A *current commit index* ι': SHA1DC hash.

The function operates as follows:

Algorithm 4.3 ($F_{\mathcal{S}}$)**.**

1. Compute $\tau' = \tau + \beta$.
2. If $\beta = 1$:

[6]We recall that SuperNova allows for a different function $F_{\mathcal{S}}$, and hence a different signing scheme $\mathcal{S}$, to be used at each step.

- Compute $\rho = A_S(\kappa_s, (\iota, \gamma), \sigma)$.
- Compute $\widetilde{\sigma} = S(\kappa_s, (\iota, \gamma), \rho)$.

Else:

- Set $\widetilde{\sigma} = \sigma$.

3. Compute $\iota' = \text{SHA1DC}(\iota, \gamma, \widetilde{\sigma})$
4. Output $s' = (\tau', \iota')$.

Fig. 1 depicts $F_{\mathcal{S}}$ in the form of a circuit. There are a number of remarks to be made about the construction, but it is convenient to first explain the intuition behind it. Essentially, the selector bit β allows the prover to choose whether they would like to feed the hash function with a hand-picked signature σ passed as part of the witness, or with a signature produced by running the signing algorithm S on a private key κ_s and message (ι, γ) (which corresponds to the commit data signed in the Git protocol, as explained in Section 2.1). In the latter case, $F_{\mathcal{S}}$ resorts to the auxiliary algorithm A_S to recover the randomness needed to produce the signature. Either way, the resulting signature $\widetilde{\sigma}$, together with the rest of the commit data, forms a commit object which is hashed in order to produce the current commit index ι'.

Separately, β is added to the counter τ, resulting in an updated counter value τ'. Consequently, if the verifier has access to ι and ι' and these differ by one, $\mathcal{V}$ knows the output index ι' was obtained by hashing, among other inputs, a signature produced with a private key the prover holds (using, as pointed out earlier, the honest signing algorithm S).

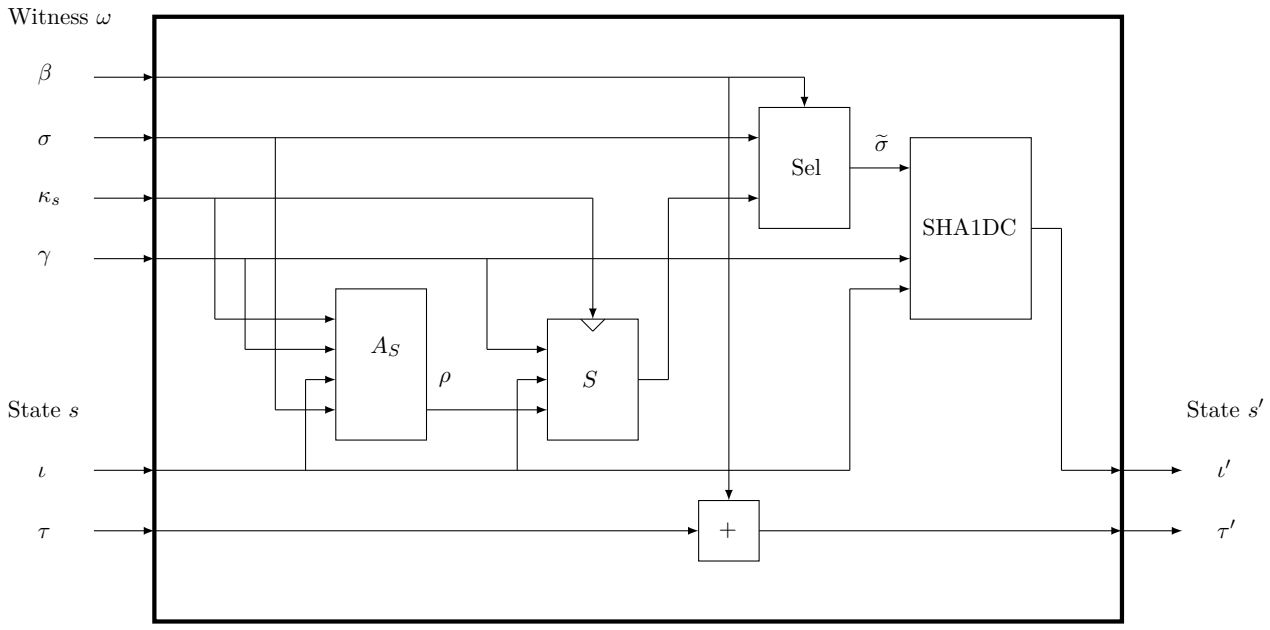

Figure 1: Design of $F_{\mathcal{S}}$ under the PoK-RS paradigm.

After n chained executions of $F_{\mathcal{S}}$, if the verifier sees τ_0 and τ_n differ by m, they know the prover supplied the private key used to sign m commits in the hash chain $\iota_1, \ldots, \iota_n$. Note that, unless $\mathcal{V}$

have access to the intermediate counters τ_i, they will be unable to pinpoint which specific steps of the commit chain those are.

Although we will delve into security considerations in a few lines, the idea underpinning this entire design is that, by the cryptographic properties of SHA1DC, if the prover is able to produce a final hash ι_n matching an honest commit index known to the verifier, then the signature $\widetilde{\sigma}$ fed to the hash function in the last execution of $F_{\mathcal{S}}$ must coincide with that of the commit in question. As for iterated execution, since the index of each commit is fed to the hash function which produces the index of the next commit, the security of the hash ensures the prover cannot produce a correct final hash ι_n using an alternate hash chain, i.e., a rogue Git history.

Remark 4.4.

1. In addition to verifying a proof of iterated execution, $\mathcal{V}$ must check that $\tau_0 = 0$ and ι_n matches the index of a public commit known to them. Alternatively, the verifier may disregard the check on τ_0 and simply consider the prover provided the signing key for $\tau_n - \tau_0$ commits.

   The matching of ι_0 against some starting known initial commit index is optional: as mentioned above, the collision-resistance properties of the hash function should ensure a dishonest prover cannot reach the correct ι_n starting at a dishonest index. A related optional check is that of n itself: the number of executions (or at least a bound on it). The significance of this is mentioned in touched upon Section 4.2.1.

2. If $F_{\mathcal{S}}$ is being applied to the root commit of a branch, the first parent commit index ι should be empty. In particular, the circuit implementation should support this case.

3. The provided signing key κ_s may be different in each execution of $F_{\mathcal{S}}$. This allows a prover who happened to sign commits with several keys in one single repository (for instance, using different GitHub accounts) to prove authorship of all of those commits.

   This is one more point in favour of the privacy-preserving focus of zkGit. By way of example, in repositories where authorship of a given number of commits with a single key would identify them (for instance, because they have authored more commits than any other user), the above design prevents this leakage of information by hiding from the verifier how many different keys were used throughout the iterated execution.

   We remark this is a feature, not a bug: if desired, it is not difficult to modify the definition of $F_{\mathcal{S}}$ to enforce the use of one single key, which can still remain unknown to the verifier.

4. The IVC scheme ensures that the prover must respect the "wiring" of the circuit. In particular, an accepted proof convinces the verifier that the stripped commit object γ fed to the signing algorithm S and to the hash function SHA1DC are the same. However, if $\beta = 1$, there is no relation between σ and $\widetilde{\sigma}$, and in particular the prover could feed A_S with any signature σ – not necessarily the one used to sign the commit. Note that this poses no security risk as A_S is only meant to allow the prover to recover the randomness used to produce the original signature, and feeding that actual is simply the most natural way to do so.

4.2.1 Security Considerations

We now devote a few lines to the security of the PoK-RS design. This discussion, although far from a full security analysis, should illustrate the main cryptographic ideas involved. In essence, we would like to argue the following:

Claim 4.5. *A verifier who accepts a Nova IVC proof of n executions of $F_{\mathcal{S}}$ with initial state $s_0 = (\tau_0, \iota_0)$ and final state $s_n = (\tau_n, \iota_n)$ can be confident that the prover holds the private keys used to sign $\tau_n - \tau_0$ commits of a publicly known n-commit chain ending on index ι_n.*

This relies on the security of three cryptographic primitives:

- The IVC scheme (Nova).
- The hash function (SHA1DC).
- The signature scheme ($\mathcal{S}$).

This order is both from outer to inner, and from simplest to most delicate. We analyse each point in turn:

The IVC scheme. The key IVC security property for our purposes is *knowledge soundness*, which essentially states that if the verifier accepts a proof as in Claim 4.5, then they can be confident that the prover has witnesses $\omega_0, \ldots, \omega_{n-1}$ satisfying Eq. (1). This is, as for most arguments of knowledge, formally defined in terms of extractors: for any (not necessarily honest) prover $\mathcal{P}^*$, there should exist an efficient extractor $\mathcal{E}$ such that the probability that $\mathcal{V}$ accepts a proof from $\mathcal{P}^*$ is negligibly larger than the probability that $\mathcal{E}$ can extract valid witnesses ω_i as above. The details can be found in [KST22], Definition 5. The Nova IVC is indeed knowledge sound (cf. op. cit. lemma 3), as is its zkSNARK variant described in sections 5 and 6 of the cited article.

Consider now the definition of $F_{\mathcal{S}}$ (for instance, as depicted in Fig. 1). The only modification of τ between one state at the next is the addition of β, which is by definition in $\{0, 1\}$ (in practice, this should be constrained). Therefore, if the ω_i are valid witnesses, one must have $\beta_i = 1$ in exactly $\tau_n - \tau_0$ of the steps. There is a small subtlety here: in an actual IVC system, τ_n would not be a signed integer but instead an element of $\mathbb{F}_p$, the field in which the arithmetic of that system takes place. Therefore, all one formally learns is that the number of steps with $\beta_i = 1$ is congruent to τ_n modulo p. However, since Git histories can have nowhere near as many as p commits for the actual primes p used in practice (in the order of 2^{256}), it is safe to choose the unique representative of τ_n in $\{0, \ldots, p-1\}$. Nonetheless, this would be an important distinction in a formal security proof. Note that, even with smaller primes, this fact would simply limit how many commit signatures can be proved, but never lead to an attack where more signatures are proved than are actually known to the prover.

Let us denote the set of these *signed indices* by $I = \{ i : 0 \leq i < n,\ \beta_i = 1 \}$. In addition to the above, a valid proof ensures that, at each step i (whether signed or not), the output index ι_i was computed correctly according to $F_{\mathcal{S}}$ (hashing included).

The hash function. Git's default hash function is SHA1DC, which forces the same choice in $F_{\mathcal{S}}$ (because the latter must produce the same commit index chain found in the actual repository). Although SHA1DC is indeed a strengthening of SHA1 preventing certain known collision attacks, it will soon become apparent that our design would benefit from a more cryptographically secure function. Migration to SHA256 has been in the works for years but is incomplete at the time of writing – most notably, due tot he lack of adoption by large platforms such as GitHub. For the sake of generality, we first replace SHA1DC by an arbitrary hash function $\mathcal{H}$ in our analysis.

Suppose a verifier accepts a proof as in Claim 4.5, in particular checking that the final index ι_n matches a known index in some repository. As discussed above, the security of the IVC scheme implies that ι_n is, with overwhelming probability, the hash of the some commit data γ_{n-1}, parent commit index ι_{n-1} and signature $\widetilde{\sigma}_{n-1}$ known to the verifier (in the notation of Fig. 1, with subindices added for clarity). If $\mathcal{H}$ is collision resistant, then these three objects must coincide with those in the actual repository: otherwise, the malicious prover would have found two different preimages of ι_n under $\mathcal{H}$.

It follows from an inductive argument that γ_i, ι_i and $\widetilde{\sigma}_i$ necessarily match their counterparts in the repository for all $0 \leq i < n$ (under the same collision-resistance assumption). Note that the argument crucially relies on the fact that each index ι_i is part of the data fed to $\mathcal{H}$ in order to produce the next one ι_{i+1}. It is due to this core relation that we termed this system "hash chain" in Section 3.

Looking into the security assumptions on $\mathcal{H}$ more closely, two potential attacks can be attempted depending on the attacker's capabilities:

- **The attacker is able to contribute one commit to the target repository**. Suppose the following:
  - The attacker has the technical knowledge required to, in preparation for the attack, make a contribution that gets accepted into the target repository. Note that this is a non-trivial assumption in many cases.
  - The attacker can find a collision $\mathcal{H}(\gamma, \iota, \sigma) = \mathcal{H}(\gamma', \iota', \sigma')$ where γ, ι and σ form the commit object of the contribution from the previous point (in particular, ι is current head of the target repository right before the contribution is made) and γ', ι', σ' are the head of a local, rogue repository containing arbitrary commits which are signed honestly by the attacker[7].

  If those assumptions hold, the attacker can claim authorship of as many commits in the remote repository as their rogue repository (although there is a natural bound, see below). To do so, they assume the prover's role and execute $F_{\mathcal{S}}$ with their rogue commits up until the point of collision, where they use γ', ι' and σ' as part of their witness, obtain the index $\iota = \mathcal{H}(\gamma, \iota, \sigma) = \mathcal{H}(\gamma', \iota', \sigma')$ of the honest repository commit, at which point they switch to the honest repository data.

  Assumption 2 requires the attacker to break a slightly harder version of the **collision resistance of $\mathcal{H}$**. Harder because the preimage components γ, ι, σ cannot be fully manipulated: ι is dictated by the honest repository (the case of the first commit in a repository requires separate treatment); γ contains a *tree index*, i.e. the hash of the actual contents of the accepted contribution, as well as other data whose manipulability is limited (committer information); and the signature, which must be computed honestly if the remote repository has active signature checks. The message, subject and date can be chosen quite freely as long as the format is preserved. These limitations should, however, not be overstated: the well-known "SHAttered"[8] attack on SHA1 is successful under similar constraints, namely structural properties of PDF documents.

- **The attacker targets a repository where they have no contributions**. If this is the case, the same attacking procedure as above could be used with the crucial difference that γ, ι and σ

[7] The rogue repository could be limited to one commit, i.e. to the commit object $(\gamma', \iota', \sigma')$, but the resulting attack would be utterly pointless: the attacker would be able to claim authorship to one commit, but they could also have done so honestly by assumption.

[8] https://shattered.io.

would be completely prescribed by the honest repository (although the attacker would free to choose which point in the commit history they want to target). Therefore, an attack of this type would need to break the **second preimage resistance** of $\mathcal{H}$, which is a harder problem than finding arbitrary collisions.

A caveat to these two attacks is that, if the verifier checks the number n of executed iterations of $F_{\mathcal{S}}$ (which they do have access to), this places a natural limit on the number of contributions an attacker can falsely claim depending on the point in the chain where the collision or second preimage was found. For instance, suppose the honest repository has 50 commits $c_1, \ldots, c_{50}$ and the attacker finds a suitable collision of $\mathcal{H}$ for the index of c_3. Then, as long as the verifier checks that the number of executions of $\mathcal{F}$ was 50, the number of falsely claimed contributions is limited to 3 by the structure of the attack and the IVC scheme itself.

It is also worth bearing in mind that, if the verifier is only interested on whether the prover made *any* contributions to the repository or not (rather than their number), the first attack above is invalidated (as performing it requires an actual contribution) and therefore only breaking the second preimage resistance of $\mathcal{H}$ leads to an attack.

Git's usual hash function $\mathcal{H} = \text{SHA1DC}$ prevents the aforementioned family of collision attacks (but not all) and is considered to be resistant against second preimages. Further research is necessary to determine the feasibility of the two proposed types of attacks, especially in terms of computational and monetary costs.

The signature scheme. Suppose that an attacker does not manage to break the security of the IVC scheme or the hash function. As already explained, the IVC ensures that, in order to claim authorship of a commit, a prover needs to set the selector bit β of the witness ω to 1 for that commit. It follows from the above analysis that this requires them to hold a signing key κ_s which, when used to sign the honest commit data (by the wiring of $F_{\mathcal{S}}$ and the security of the hash), produces the same signature found in the honest repository. This should be hard to do if the prover does not hold the actual key which was used to sign the honest commit – even when they know the signatures of other messages under that same key. We capture this hardness in the following security definition:

Attack Game 4.6. Let $\mathcal{S} = (G, S, V)$ be a signature scheme. The challenger $\mathcal{C}$ and arbitrary adversary $\mathcal{A}$ engage in the following game:

- $\mathcal{C}$ generates $(\kappa_s, \kappa_v) \leftarrow G$ and sends κ_v to $\mathcal{A}$.
- For $1 \leq i \leq Q$, $\mathcal{A}$ queries $\mathcal{C}$ with a message μ_i and receives back its signature $\sigma_i \leftarrow S(\kappa_s, \mu_i)$.
- At the end, $\mathcal{A}$ outputs a message μ and signing key κ.

$\mathcal{A}$ is said to win the game if $\mu \notin \{ \mu_1, \ldots, \mu_Q \}$ and $S(\kappa, \mu) = S(\kappa_s, \mu)$.

Note that this is a relatively conservative model of our situation of interest: the game gives the adversary complete control over the messages signed by the challenger. This represents, for instance, a situation where a would-be zkGit cheater could influence someone else's signed commit data (for instance, because they have the power to choose or modify that person's commit messages). Furthermore, the adversary is *adaptive*: they are required to choose which message they will re-sign only after seeing the signatures of other messages.

Definition 4.7 (Existential Irreproducibility under (adaptive) Chosen Message Attack (EIR-CMA))**.** *In the notation of Attack Game 4.6, the re-signing advantage of $\mathcal{A}$ against $\mathcal{S}$ is*

$$\mathrm{Adv}^{\mathrm{Q}}_{\mathrm{EIR-CMA}}[\mathcal{A};\mathcal{S}] = \Pr[\mathcal{A} \textit{ wins}].$$

$\mathcal{S}$ is said to be EIR-CMA secure *if $\mathrm{Adv}^{\mathrm{Q}}_{\mathrm{EIR-CMA}}[\mathcal{A};\mathcal{S}]$ is negligible for all PPT adversaries $\mathcal{A}$ and poly-bounded Q (as usual, in some implicit security parameter λ).*

This security definition is of our own design and not a standard one. In fact, it as far as the authors are aware, it does not seem to be directly implied by any standard security definition of signing schemes. The reason is simple: in most real-world usages of signing schemes, a malicious adversary is not bound to use the honest signing algorithm $\mathcal{S}$ when producing signatures – and should not be required to do so in a security game. In our case, however, this is guaranteed by the provable execution of $F_{\mathcal{S}}$.

A natural comparison can drawn to the definition of *existential unforgeability under an adaptive chosen message attack* (EUF-CMA) [GMR88], which is a strong but fairly standard one. Neither is it stronger than Definition 4.7 (for the reason outlined above), nor is it weaker, as the key κ recovered in Attack Game 4.6 may not coincide with κ_s and therefore cannot be used to produce signatures that verify under κ_v.

Nonetheless, it is possible to prove the EIR-CMA security of OpenPGP's signature schemes. Indeed, we can recycle the proof techniques used in the proof of EUF-CMA security of RSA-FDH signature scheme to show its EIR-CMA security. A careful analysis of the OpenPGP schemes is necessary, and the PoK-RS design should only be applied whenever $\mathcal{S}$ satisfies EIR-CMA security.

4.3 Proof of Knowledge through Key-pair Relation

We now briefly touch upon an alternative paradigm to the one described above, which we have termed "proof of knowledge through key-pair relation" (PoK-KPR). It hinges on the following notion:

$\mathcal{P}$ can demonstrate they hold the signing key by proving it forms a valid key pair with a verification key κ_v which verifies the original signature.

The Assumption 4.2 on the signature scheme $\mathcal{S}$ is no longer necessary. Instead, one must be able to efficiently check whether two keys form a valid key pair for the signature scheme $\mathcal{S} = (G, S, V)$:

Assumption 4.8 (Relatability Assumption)**.** There exists an efficient deterministic algorithm $R_G = R_G(\kappa_s, \kappa_v)$ which, given a signing key κ_s and a verification key κ_v, outputs 1 if

$$\Pr[(\kappa_s, \kappa_v) \leftarrow G] > 0$$

and 0 otherwise.

For example, algorithm R_G for DSA signature scheme would take $\kappa_s = (g, y)$ and $\kappa_v = x$, and check that $g^x = y$ (which must be checked using circuit arithmetic in some predetermined $\mathbb{F}_p$). Note that $\Pr[G \text{ outputs } (\kappa_s, \kappa_v)] > 0$ may be a stronger condition than "(κ_s, κ_v) is a valid key pair", but the abstract definition of a signature scheme does not have an intrinsic notion of validity other than that. Although we will not go into the same level of detail as with PoK-RS, the design of the function F for PoK-KPR can be summarized as follows:

- All inputs and outputs of $F_{\mathcal{S}}$ (cf. page 17 f.) remain except for β, which disappears. A verification key κ_v is added to the witness ω.
- In Fig. 1, the auxiliary and signing circuits A_S and F are replaced by a verification circuit V. The latter receives the signed data (γ and ι), signature σ and verification key κ_v and outputs a validity bit υ indicating whether the signature is valid.
- A circuit implementing the algorithm R_G from Assumption 4.8 is added to the circuit, receiving κ_s and κ_v and outputting a relationship bit ε as defined.
- The SHA1DC circuit remains unchanged, but now directly receives the signature σ rather than $\widetilde{\sigma}$ (which is no longer defined).
- Crucially, the relation $\tau' = \tau + \beta$ from Fig. 1 is replaced by $\tau' = \tau + (\upsilon \cdot \varepsilon)$.

Intuitively, we have swapped the bit β from the PoK-RS, which captured the notion that "the prover has signed the data with a signing key they hold", by the bit $(\upsilon \cdot \varepsilon)$, which conveys that "the prover holds a signing key matching a verification key which correctly verifies the data". We refer the reader to Remark 4.4, a number of whose points also apply to the PoK-KPR design.

The security considerations around this design are the same as for PoK-RS as far as the IVC and hash function are concerned. In particular, the same two possible attacks on the hash function are possible here. For the signature scheme, the following informal security definition should replace EIR-CMA security (Definition 4.7): for any key pair $(\kappa_s, \kappa_v) \leftarrow G$ and any PPT adversary that queries the challenger for signatures under κ_s (and receives κ_v), the probability of outputting $(\mu, (\widetilde{\kappa_s}, \widetilde{\kappa_v}))$ such that $1 \leftarrow V(\widetilde{\kappa_v}, \mu, S(\kappa_s, \mu))$ and $1 \leftarrow R_G(\widetilde{\kappa_s}, \widetilde{\kappa_v})$ is negligible.

In comparison to PoK-RS, the PoK-KPR approach:

- Relies on a security assumption which is closer to standard ones for signing schemes.
- Relies on the Assumption 4.8 and the in-circuit implementation of the relation algorithm R_G.
- Does not rely on the Assumption 4.2 or the in-circuit implementation of the randomness-recovering algorithm A_S.
- Might be more or less computationally expensive, depending on the cost of verification as opposed to signing in scheme $\mathcal{S}$, as well as the costs of R_G versus A_S.

5 Merge policies

Throughout this document, our aim has been to construct a protocol for proving knowledge of the secret key used to sign commits in a given repository. Unfortunately, this does not always allow one to prove contributions in arbitrary repositories. In fact:

It is not always possible to prove contributions to a Git repository anonymously relying solely on data from the commit chain.

The goal of this section is to explain this claim, how it relates to the design from Sections 3 and 4, under which assumptions the same design can be maintained without modification, and which changes can be made to prove authorship in repositories satisfying certain assumptions.

There are two levels to the issue. On the most superficial one, complications to the above design arise when the contributor is not the same person who merges the commit into the main branch. This is very frequently the case in large repositories, which are managed by a few select maintainers but receive contributions from a much larger set of users. The party who signs a commit (if anyone) is its *committer*—essentially, the person calling the `git merge` (or similar) command. This may or may not coincide with the *author* of the commit, the person who authored the code (i.e., the contributor). For completeness, we mention two different scenarios:

- If a maintainer runs `git merge` locally, they will be the committer of the newly formed commit which is added to the current branch, whereas the author of the commit being merged into it will appear as the author of the new commit.
- If a maintainer merges a commit using the GitHub interface (typically from a PR), the author is handled in the same way as above, but the committer is GitHub itself (with the identity `GitHub <noreply@github.com>` and public key with ID `4AEE18F83AFDEB23`) and the maintainer simply appears as a *co-author* mentioned in the commit message. Note that the latter is not a distinguished field of the Git protocol, but rather a GitHub convention.

As an example of the latter, consider the commit `c02d43f30174d923cfe3d2f73e29911160658353` in `arkworks-rs/crypto-primitives`. As can be seen on GitHub, this is the commit merging PR #60. This PR, authored by user Tom Shen (`tsunrise`), was hosted in a different GitHub repository (fork) belonging to that user and contained commits signed by them. However, the commit which merged this PR onto the `arkworks` repository only contains the following (part of the message has been omitted):

```
$ git cat-file -p c02d43f30174d923cfe3d2f73e29911160658353
tree 8c0593fd1d1e0a2d892fd9cd937815e0371dd323
parent 3402a729b7104983c9779714ab8580de5187bbdd
author Tom Shen <tomshen@berkeley.edu> 1626331687 -0700
committer GitHub <noreply@github.com> 1626331687 -0700
gpgsig -----BEGIN PGP SIGNATURE-----

 wsBcBAABCAAQBQJg79onCRBK7hj4Ov3rIwAASV8IAKPwYekZ61omlZnSbf+bWRrn
 dUsMerOW/bS2r/2jmKpvylXfLiBkilSWMWnhVSAG9YEEKurKIDaMTRm0IncP0qex
 AoVCTXwB7Hu2r6/j7cANnhb9J4sjdc0t4xeEXFQBUPR+CkVSL/MeQBmIVfiVPAjS
 sheBJN8+TzKsjvX4nIG9/X9/+rZaT8XUsuEmfIsKPZsknHT2Cp/EB96CmXBV8eRB
 H/MRSHMYqCk0e71/jpPvNdz9c55+jjNFXPmjjLcr4WhH4EEFX1idma76MJFdJuRX
 tG73e3iSKTnPi3/Pb1O73HeAAYvgkfvzrOFbCa3pIx4Rlh1KYbsuvcbJ2D8X0SM=
 =IbHS
 -----END PGP SIGNATURE-----


Merkle Tree v3 & Update Poseidon (#60)

(...)

Co-authored-by:  weikeng <w.k@berkeley.edu>%
```

Whether merging commits locally or on GitHub, the above explanation makes it clear that proving knowledge of a private key used to signed a commit on the main branch is not always possible for a contributor. However, authorship can still be traced using signatures if the commit in question is a *merge commit*. As an example, consider `https://github.com/ubuntu/gnome-sdk`. The commit `db3d3c1637fcbe6fc16922675ff76bd8b3b500b8`, merging PR #174, has two parents: it is a merge commit in the Git-protocol sense, and the second parent index corresponds to a commit whose committer (and therefore signer) is the actual contributor.

The second, deeper level of the issue we are dealing with is the following: merging branches does not always produce merge commits. There can be various reasons for this, of which we illustrate a couple:

- If the head of the current branch is an ancestor of the commit we are trying to merge into it, `git merge` will instead *fast forward* the current branch, appending the newer commits to the previous state. This is not a problem *per se*, as it would actually preserve the contributor's signatures as part of the main chain. Note that fast-forwarding can be prevented (even in cases where it could be applied) using Git's '–no-fast-forward' option, which always results in merge commits.

- If several contribution commits are *squashed* into one to be merged into the current branch, there is no hope for the signature-based approach: the original contributor's signatures are lost and the new commit is signed by the committer (whether a maintainer or, e.g. GitHub on their behalf).

  When this happens, the resulting commit can only be related to the actual contributor through the explicit "`author Name <email>`" field, and not through signatures. This implies that the only way a verifier can be convinced about authorship of contributions using only commit data from GitHub necessarily involves them learning the name and email of the contributor.

- If contributions are merged into the current branch through *rebasing*, the same fundamental issue arises as in the previous point.

Unfortunately, squashing and rebasing are two ubiquitous policies, especially on platforms such as GitHub. This means the PoK-RS and PoK-KPR strategies are only applicable to repositories with specific merge configurations. We now explain three (and a half) possible approaches depending on the specifics of the repository of interest:

Unmodified protocol for repositories where contributors are committers (anonymous) As explained above, in any repository where the person making each contribution is also the committer for that contribution, the protocol from the previous two sections can be used without modification. This workflow is admittedly not too common in the kinds of repositories where zkGit would be of interest. By anonymous we mean that authorship is proved using PoK of a key, rather than explicitly revealing author information.

Modified protocol for repositories containing merge commits (anonymous) In a merge commit, signature traceability is still maintained in the form of the second parent index in the commit object. The following modification of the PoK-RS design maintains the desired functionality: starting from the original construction of $F_{\mathcal{S}}$ (Fig. 1),

- add to the witness ω fields for commit data coming from a different branch: $\widetilde{\gamma}, \widetilde{\iota}, \widetilde{\sigma}$ (the old intermediate variable $\widetilde{\sigma}$ disappears).
- let the signature-related components of $F_{\mathcal{S}}$ (A_S and S) act on this branch commit data (with parent $\widetilde{\iota}$) rather than the data γ, ι, σ from the main branch.
- add a second SHA1DC chip for the branch commit data.
- add to the witness a *second parent index* $\iota^{(2)}$ (which could be empty, e.g. in non-merge commits).
- add to the witness a selector $\pi \in \{0, 1\}$ controlling whether the the second parent index fed to the original hashing chip (which hashes main-branch commit data) should be $\iota^{(2)}$ or the output of the new hashing chip.
- remove the selector bit β. Instead, it is π that is added to the counter: $\tau' = \tau + \pi$.

Figure 2 shows a simplified diagram of the new design (for instance, the counter is omitted for visual clarity). The idea of the resulting function is simple: whenever the prover knows the key κ_s used to sign the contributed commit which was merged onto the main commit chain, they set $\pi = 1$ and use that commit data to produce the second-parent index appearing in the main-chain commit. If the hash function is secure, it this is impossible to do so using dishonest data for the reasons explained in Section 4.2.1.

Other variants of this approach can easily be constructed: for instance, one can easily add support for octopus merges (where more than one branch is merged into the main one at once). Furthermore, it is easy to swap the PoK-RS design of the new $F_{\mathcal{S}}$ by PoK-KPR (cf. Section 4.3).

Moderately modified protocol for repositories using `git merge` (anonymous) It is not difficult to define a function $F_{\mathcal{S}}$ which combines the previous two approaches, allowing contributions in the form of signed commits on *either* the main chain (as in the original design) or a branch that was merged in a merge commit (as in the second one). This covers the two scenarios that can occur when using `git merge`: fast forwarding and genuine merging. Therefore, if that command is consistently used to incorporate contributions throughout a repository, the IVC PoK paradigm can still be used. We stress that the resulting circuit relies exclusively on the knowledge of a signing key to prove authorship, thus avoiding revealing explicit contributor data (name and email) to the verifier.

Heavily modified protocol for repositories where contributor signatures are lost (not anonymous) When policies such as squashing and rebasing are in place, the only way[9] to prove authorship is by explicitly letting the verifier know the contributor's identity (name and/or email). For such situations, we propose the following modification of $F_{\mathcal{S}}$, which still maintains the IVC paradigm from Section 4. Starting again from the original construction of $F_{\mathcal{S}}$ (Fig. 1),

- add to the witness ω a private author field α_ω (representing name and email). Now the original committer data field γ in the witness is no longer meant to contain author information.
- add to the public input z a public author α_z (ibidem).

[9] At least if one relies only on Git data, and not on application-specific features such as the GitHub API.

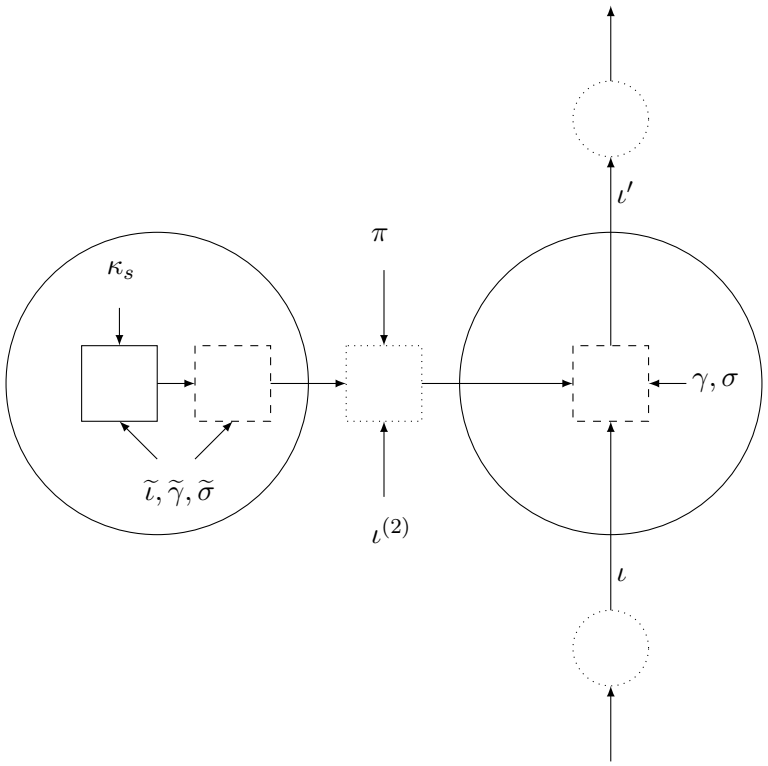

Figure 2: A modification of the PoK-RS design of the function $F_{\mathcal{S}}$ for merge commits (some details are omitted). The dotted circles and the solid circle to the right represent the repository's main commit chain. The two solid circles correspond to a single execution of the new $F_{\mathcal{S}}$, with the one to the left containing the commit data of the branch being merged—which in turn contains the contributor's signature. The solid square is the signing chip implementing S, the dashed rectangles are hashing chips implementing SHA1DC and the dotted square is a selector controlled by π.

- add to the public output z' an author $\alpha_{z'}$, which is directly wired to α_z.
- remove the signature-related components A_S and S. The witness signature σ is directly fed to the hash function.
- make the selector bit β control whether α_ω or α_z is fed to the hash function (as opposed to choosing between two signatures, one of which no longer exists). The counter τ is still incremented with β at each iteration.

When the verifier receives an IVC proof with $F_{\mathcal{S}}$ as above, they check the initial and final counters $\tau_0 = 0$ and τ_n, as well as the final index ι_n, as before (optionally, n and the initial index can be matched against known data). If the proof verifies correctly, then they can be confident that the author $\alpha_0 = \alpha_n$ (which they see in the public instance) indeed appears as the author of τ_n commits.

The implementation of the above modification must be done carefully, especially in terms of correctly hashing the various fields into a commit index. A natural attack that the implementation

should prevent would have an attacker managing to get their user data as the first part of the subject of a commit contributed by someone else. If this were the case, setting $\beta = 1$ and removing their data from the commit subject (which is now the initial part of the stripped commit data γ) would produce a valid input to the hash function and therefore allow the attacker to falsely claim authorship of that commit. Indeed, similar attacks could also be devised on the original design of $F_{\mathcal{S}}$, although those would be harder to perform. Nonetheless, such possibilities should be prevented at the implementation level and depend on the specifics of the IVC and hashing libraries used.

On a higher level, this protocol loses anonymity and in that sense takes us back to systems $\varnothing$ and B from Section 3. Unlike those, however, it only requires the storage of one piece of data in addition to the user-identifying data (verification key before, name and email now): the head index in the (branch and) repository of interest. It preserves the core idea of the index chain and therefore still takes advantage of the IVC paradigm used in system A. This is as opposed to having to store all the signatures of a potentially large commit history. Nonetheless, realistic usage cases where this modification of system A would be advantageous over systems $\varnothing$ and B need to be determined.

Proof submission We conclude this section by briefly addressing a point which has been tacitly ignored up until now. In the traditional approach to proving authorship, digitally signed documents, an author signs some data with a private signing key and any interested party can verify the signature using the author's public verification key. The authorship claim is inextricably linked to this verification key.

By contrast, in an argument of knowledge, a verifier is simply convinced that a prover knows a witness satisfying a certain relation with the public instance. In interactive settings, the upshot is simple: $\mathcal{V}$ learns this about the party *they are interacting with*. In non-interactive settings, however, issues such as proof cloning arise: Suppose a genuine contributor generates a zkGit proof, but a malicious prover gets hold of it and submits it as their own. This could easily be done through a man-in-the-middle attack or by monitoring the author's network traffic. A more devious attack would use proofs submitted to, say, a false funding call in subsequent, applications.

This means that, in many settings, a way to bind a proof to its author is required—and this should ideally not be done at the expense of anonymity. The resulting augmented proofs should also not be *malleable* in the sense that, by retrieving one such proof, one could easily create an augmented proof with different author information.

We will not delve into the above question in this document and instead defer it to a later stage of the project. It is however worth pointing out that this challenge is far from unique to zkGit and has been explored extensively, for example in the context of anonymous payment networks such as ZCash [Zca] or Tornado Cash [Tor].

6 Conclusions

Proving Git contributions under certain succinctness and anonymity restrictions is an interesting cryptographic question with potential real-world applications. Git commit signatures as proofs of authorship fit nicely with new developments in the theory of zkSNARKs and provide a good starting point.

The recently introduced concept of incrementally verifiable computation, which allows a prover to convince a verifier about the output of chained execution of a function, presents itself as an excellent tool due to the linked nature of commit indices in Git repositories. Specifically, constructing a

function F whose public input/output represents a commit index and a signature counter, and which takes as additional private inputs certain signing data, a verifier can be convinced about the prover's correct recreation of a given commit history (with signatures at some of its steps) by just holding the final commit index in it. Alternatively, one can opt for a system where the verifier is required to hold the signatures of all commits in the branch of interest and the prover simply performs the verification for for a specific public key known to the verifier on their behalf. However, this approach lends itself very poorly to on-chain deployment and loses the desired anonymity.

In order to convince the verifier about the knowledge of the signing key used to sign a commit in the repository of interest, the prover can either use an F which re-signs that commit (PoK-RS) or one that checks the validity of a supplied key pair and verifies the commit signature with the verification key therein (PoK-KPR). This results in two possible designs of F with different functionality and security requirements.

The Nova article [KST22] leverages the concept of *folding schemes* to define an IVC scheme where the verifier's work is independent of the number of steps to be proved and therefore, in the case of zkGit, of the number of commits in the repository of interest. Its recent generalisation, SuperNova, allows for the use of a different function F at each step of the computation, which would enable support for poofs containing different signing schemes $\mathcal{S}_1, \ldots, \mathcal{S}_t$ by constructing separate functions $F_{\mathcal{S}_1}, \ldots, F_{\mathcal{S}_t}$.

Under a preliminary security analysis, the above approach provides a satisfactory solution to the zkGit question whenever the contributor is the committer to the branch of interest—and, therefore, the party signing the commit. However, many Git repositories handle contributions differently, often relying on maintainers (different from the authors) to bring contributions in. A minor modification of our original design uses non-first parent indices to provide the desired functionality whenever contributions are merged into the branch of interest using traditional Git commands such as `git merge`. However, merge policies such as squashing and rebasing destroy contributor signature information in a way that leaves as the unique reference to the contributor its user data inside the *author* field. Although a way to adapt our design to this situation is proposed, the result is unsatisfactory in that the verifier learns the identity of the contributor. The bottomline is that, for such repositories, anonymous proving of contributions is not possible—at least, relying solely on Git data.

The natural next step is the evaluation of the extent of the aforementioned limitations, and therefore, of the usefulness of our main design. If traditional Git merging is widespread enough in repositories where zkGit would be of interest, then a more thorough security analysis should follow, together with a case-by-case study of OpenPGP signature schemes to decide which of PoK-RS and PoK-KPR is more suitable for each. Implementation would be a challenging, yet fascinating task.

References

[AW18] Andreas M Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'reilly Media, 2018. URL: https://github.com/ethereumbook/ethereumbook.

[Cli] *Clique*. https://www.clique.social/. Accessed: 2023-11-21.

[Dec] *Deco*. https://www.deco.works/. Accessed: 2023-11-21.

[Ebe21] Jacob Eberhardt. "Scalable and Privacy-preserving Off-chain Computations". PhD thesis. Technischen Universität Berlin, 2021. URL: https://api-depositonce.tu-berlin.de/server/api/core/bitstreams/2b81beb7-5b0f-4048-a56f-104317a82675/content.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks". In: *SIAM Journal on Computing* 17.2 (1988), pp. 281–308. DOI: 10.1137/0217017. eprint: https://doi.org/10.1137/0217017. URL: https://doi.org/10.1137/0217017.

[Git] *Git source code on GitHub*. https://github.com/git/git/blob/3b3d9ea6/sha1dc/sha1.c#L23. Accessed: 2023-11-27.

[KS22] Abhiram Kothapalli and Srinath Setty. *SuperNova: Proving universal machine executions without universal circuits*. Cryptology ePrint Archive, Paper 2022/1758. https://eprint.iacr.org/2022/1758. 2022. URL: https://eprint.iacr.org/2022/1758.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: *Advances in Cryptology – CRYPTO 2022*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Cham: Springer Nature Switzerland, 2022, pp. 359–388. ISBN: 978-3-031-15985-5.

[Rec] *Reclaim*. https://drive.google.com/file/d/1wmfdtIGPaN9uJBI1DHqN903tP9c_aTG2/view. Accessed: 2023-11-21.

[SBKAM17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. "The First Collision for Full SHA-1". In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 570–596. ISBN: 978-3-319-63688-7.

[Sta] *StackOverflow*. https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/. Accessed: 2023-11-09.

[Tls] *TLS Notary*. https://tlsnotary.org/. Accessed: 2023-11-26.

[Tor] *Tornado Cash*. ipns://tornadocash.eth/. Accessed: 2023-11-27.

[Tow] *Town Crier*. https://www.town-crier.org/. Accessed: 2023-11-21.

[Val08] Paul Valiant. "Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency". In: *Theory of Cryptography*. Ed. by Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–18. ISBN: 978-3-540-78524-8.

[ZCCJS16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. "Town Crier: An Authenticated Data Feed for Smart Contracts". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 270–282. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978326. URL: http://doi.acm.org/10.1145/2976749.2978326.

[ZMMGJ20] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. "DECO: Liberating Web Data Using Decentralized Oracles for TLS". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. Virtual Event, USA: Association for Computing Machinery, 2020, 1919–1938. ISBN: 9781450370899. DOI: 10.1145/3372297.3417239. URL: https://doi.org/10.1145/3372297.3417239.

[Zca] *ZCash*. https://z.cash/. Accessed: 2023-11-27.