Leveraging Duality for Programming with zkSNARKs

Rahul Krishnan
University of Wisconsin–Madison
rahulk@cs.wisc.edu

Ethan Cecchetti
University of Wisconsin–Madison
cecchetti@wisc.edu

Extended Abstract

Non-Interactive Zero-Knowledge (NIZK) Proofs enable a prover to cryptographically prove, for any polynomial-time program p, that p evaluates to y on inputs $\overline{x}$ while hiding any combination of inputs and outputs. NIZK proofs provide an immensely powerful abstraction, but for years were only practical in bespoke cryptographic protocols. In the last decade, however, the rise of practical Succinct Non-interactive Arguments of Knowledge (zkSNARKs) has enabled numerous academic and industrial applications [e.g., 7, 8, 10–13, 15].

Efficient proofs often require a *witness* that a claim holds. The process of constructing this witness and verifying the proof are extremely similar, yet current tools for specifying zkSNARKs fail to address this *inherent duality*. They instead require two separate pieces of code that are written independently, often live in separate parts of an application, and must be kept in sync, leading to maintenance nightmares.

For example, consider data inclusion proofs for Merkle trees, an authenticated data structure used in numerous protocols [e.g., 3, 5, 9, 14]. A Merkle tree is a binary tree with data associated with each leaf. Each node is given a value by hashing the values of its two children (or the value of the data for a leaf node). The "root hash" value assigned to the root serves as an identifier for the whole tree and supports efficient inclusion proofs, among other operations. To prove inclusion of a value d, one can iteratively hash d together with the sibling of each node on the path from d to the root and then check that the result matches the root hash. Generating such a proof requires computing the list of sibling hashes—the witness—and iteratively hashing up the tree, two very similar operations.

We present a new language feature, a `compute_and_prove` block, that binds a new variable representing the witness, and contains a single piece of code that both computes the value of the witness and proves the witness is valid. The block then returns a cryptographic proof object that other parties in the system can independently verify.

Figure 1 shows how to use this feature to specify a Merkle inclusion proof where the root hash of the tree is `ROOT_HASH`, the hash of the data is `LEAF_HASH`, and the leaf node containing that data is `LEAF`. The code loops over each level of the tree, going left or right as specified by `PATH`—the path specified leaf-to-root. Line 12 places the value of the current node's sibling in `sib_hashes`, computing the current level of the witness. Lines 14–16 use that value and the running hash to compute the next value. Finally, line 19 checks that the computed root hash matches the original root hash. Instead of two separate code pieces, this block contains a single loop (lines 5–17), reusing structural elements, and clearly connects the definition of the witness (line 12) to its use (lines 14–16).

```
 1 proof pf = compute_and_prove(witness sib_hashes) {
 2   // Initialize hashes and current node to leaves
 3   value = LEAF_HASH;
 4   next_node = LEAF;
 5   for(i = 0; i < depth; i++) {
 6     // Update the next node
 7     next_node = next_node.parent;
 8     // Get the current node's sibling
 9     sibling = (PATH[i] == LEFT ? next_node.right
10                                : next_node.left);
11     // Set sibling hash (witness computation)
12     sib_hashes[i] = sibling.hash_value;
13     // Update the hash value (witness use)
14     value = (PATH[i] == LEFT
15               ? hash(value, sib_hashes[i])
16               : hash(sib_hashes[i], value));
17   }
18   // Verify root hash matches what is expected
19   assert value == ROOT_HASH;
20 }
```

Figure 1. Merkle Proof Combined Code

To connect to existing tools for compiling zkSNARKs [2, 4, 6], `compute_and_prove` blocks compile to a lower-level language with `prove` and `verify` primitives, representing basic cryptographic operations of NIZK proofs. A `prove` statement creates a cryptographic proof object proving that some expression evaluates to true on the supplied inputs—including any witnesses that must be computed separately—while hiding a subset of those inputs, and `verify` checks the validity of that object.[1] A `compute_and_prove` statement represents both constructing the witnesses and the proof together.

Projecting Combined Code. Performing the computation requires determining which parts of the code belong to which operation. We support this separation by adding a label ℓ to the type of each variable inside a `compute_and_prove` block, indicating which operations utilize them. The possible labels, `C`, `P`, and `CP`, encompass the possible options: computation, proof, or both, respectively.

[1]The `verify` construct must already be available to verify the results of `compute_and_prove` blocks.

To separate a `compute_and_prove` block into its corresponding pieces, we define a projection operation $[\![\cdot]\!]_\ell$ that extracts the code for witness computation or proof generation, for $\ell = \mathtt{C}$ and $\ell = \mathtt{P}$, respectively. Projection uses the type labels to determine which expressions to project in each case, including $\mathtt{CP}$ expressions in both. The full translation for a `compute_and_prove` block with expression e first defines the witness variables, then runs $[\![e]\!]_\mathtt{C}$ to assign these witnesses, and finally uses a `prove` statement over $[\![e]\!]_\mathtt{P}$ to construct the output proof.

To simplify defining and referencing them, the $\mathtt{C}$-projection encodes witness variables as mutable references. However, other parties must be able to execute the proof code to verify a proof, so it cannot directly use mutable references it does not create. We therefore project a witness variable x differently in the two projections.

$$[\![x]\!]_\ell = \begin{cases} \mathrm{deref}(x) & \text{if } \ell = \mathtt{C} \\ x & \text{if } \ell = \mathtt{P} \end{cases}$$

To differentiate assignment to witness from assignment to other references, we introduce a new syntactic form $x \leftarrow e$. The type system requires x to be a witness variable and e to be available in the witness computation. The expression then projects as follows.

$$[\![x \leftarrow e]\!]_\ell = \begin{cases} x \coloneqq [\![e]\!]_\mathtt{C} & \text{if } \ell = \mathtt{C} \\ () & \text{if } \ell = \mathtt{P} \end{cases}$$

That is, in witness computation it becomes an assignment, while in proof generation it disappears entirely because it is not part of that operation.

Other expressions are not inherently limited to one of witness computation and proof generation. Projection thus relies on the type labels of variables to determine whether to project an expression to the specified part of the computation. An included expression projects to the same operation, with recursively projected subterms. An excluded expression projects to unit.

Semantics. The compilation process using projection as described above gives `compute_and_prove` blocks a semantics by translation to a lower-level language using the `prove` operation. This semantics precisely describes how a compiler should behave on our new language construct, but it performs a highly non-local transformation on the body of `compute_and_prove` blocks, making it extremely difficult for programmers to reason about. To make reasoning about the source code directly simpler, we also include a second semantics that directly describes the execution of the combined program.

Because this second semantics is meant to simplify reasoning, it is critical that it behave identically to the projection-based semantics of a compiler. Our ongoing work therefore aims to prove the following powerful adequacy theorem. Here we write $e \longrightarrow e'$ to denote operational semantic steps in the low-level language, $e \twoheadrightarrow e'$ for steps in the combined language without first compiling, and $\mathcal{F}[\![e]\!]$ to denote the full compilation described above, using both projections.

Theorem 1 (Projection Preserves Semantics (Adequacy))**.** *Suppose that* $\Gamma \vdash \mathtt{compute\_and\_prove}(w : \tau)\{e\} : \mathtt{proof}$*. Then for any store* σ*,*

$$\langle \mathcal{F}[\![\mathtt{compute\_and\_prove}(w : \tau)\{e\}]\!] \mid \sigma\rangle \longrightarrow^* \langle v \mid \sigma'\rangle$$
$$\Updownarrow$$
$$\langle \mathtt{compute\_and\_prove}(w : \tau)\{e\} \mid \sigma\rangle \twoheadrightarrow^* \langle v \mid \sigma'\rangle$$

While this theorem says the source and compiled programs produce identical results, we also aim to prove that the visible impacts of the computation itself are the same, meaning the programs are deeply identical. To formulate that statement, we look to language of robust compilation, and demand that $\mathcal{F}[\![\cdot]\!]$ satisfy *Robust Relational Hyperproperty Preservation*, the strongest correctness condition in the robust compilation hierarchy [1].

A major challenge we are still addressing is how to prove these theorems. First note that our compiler-based semantics places all witness assignments in the computation phase, while some uses will remain in the proof phase, which executes later. The in-order semantics of the surface language will necessarily behave differently if the proof code references a witness *before* it is later (re)assigned. We therefore restrict how code can interleave witness assignments and references. Specifically, our type system tracks both and requires that each witness variable is assigned before it is used, and never after.

Even with this added restriction, proving Theorem 1 will not be easy. Adequacy theorems are typically proven using a lock-step bisimulation argument. That is, for each step taken in one semantics, there is one or more corresponding steps in the other. The structure of our $\mathcal{F}[\![\cdot]\!]$ compilation makes such an argument impossible; the witness computation code executes in its entirety before the proof generation code begins. As these theorems are critical to proving the validity of our dual-semantics, we are currently investigating alternative proof strategies.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium (CSF '19)*. https://doi.org/10.1109/CSF.2019.00025

[2] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz Tapia, Albert Rubio, and Jordi Baylina. 2023. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2023), 4733–4751. https://doi.org/10.1109/TDSC.2022.3232813

[3] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *35th IEEE Symposium on Security and Privacy (S&P '14)*. https://doi.org/10.1109/SP.2014.36

[4] Cairo developers. 2024. Cairo v2.8. https://www.cairo-lang.org/. Accessed November 2024.

[5] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *26th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '17)*. https://doi.org/10.1145/3132747.3132757

[6] Gnark developers. 2024. `gnark` zk-SNARK library. https://github.com/ConsenSys/gnark. Accessed November 2024.

[7] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. 2023. Efficient Proofs of Software Exploitability for Real-world Processors. *Proceedings on Privacy Enhancing Technologies (PETS)* 2023 (2023), 627–640. Issue 1. https://doi.org/10.56553/popets-2023-0036

[8] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. 2022. Zero-Knowledge Middleboxes. In *31st USENIX Security Symposium (USENIX Security '22)*. https://www.usenix.org/conference/usenixsecurity22/presentation/grubbs

[9] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf.

[10] Michael Rosenberg, Mary Maller, and Ian Miers. 2022. SNARKBlock: Federated Anonymous Blocklisting from Hidden Common Input Aggregate Proofs. In *43rd IEEE Symposium on Security and Privacy (S&P '22)*. https://doi.org/10.1109/SP46214.2022.9833656

[11] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. 2023. zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure. In *44th IEEE Symposium on Security and Privacy (S&P '23)*. https://doi.org/10.1109/SP46215.2023.10179430

[12] Nitin Singh, Pankaj Dayama, and Vinayaka Pandit. 2022. Zero Knowledge Proofs Towards Verifiable Decentralized AI Pipelines. In *26th Financial Cryptography and Data Security (FC '22)*. https://doi.org/10.1007/978-3-031-18283-9_12

[13] StarkWare Industries. 2018. StarkWare. https://starkware.co/. Accessed November 2024.

[14] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).

[15] Zcash Foundation. 2016. Zcash. https://z.cash. Accessed November 2024.