The Siren Song of zkVMs: Lessons from a Failed Project to Build zk-SNARKs for Video Editing

Alexander (Sasha) Frolov

University of Maryland, College Park, MD, USA
sfrolov@umd.edu

Abstract. Building zero-knowledge proof systems for image/video editing is a common research direction in the current applied cryptography/zk-SNARK literature. From 2024 to 2026, my collaborators and I worked on a project to implement video editing in zk-SNARKs. For various reasons, we initially chose to implement the experiments for our paper using a *zero-knowledge Virtual Machine*, or zkVM. We spent about a year working on this version of the project. The zkVM-based system we built for our paper's initial submission ended up being $\sim$240$\times$ slower than the concurrent state of the art, and this submission was rejected.

For the second iteration of our paper, we switched to using the Noir/Barretenberg programming environment for zk-SNARKs. This system was an order of magnitude slower than the state of the art, and our second submission was also rejected.

Our final, publicly posted version of the paper switched to an implementation based on a modified folding-based proof system.

I discuss a few major lessons from this project. First, I argue that while "beginner-friendly" libraries for working with zk-SNARKs may seem like promising research tools, they have many drawbacks, some of which may not be immediately obvious. Second, I discuss some lessons around research best practices from this project. Last, I discuss some difficulties around making fair comparisons among prior zk-SNARK systems for proving image/video editing which may not be obvious from a cursory reading of the relevant papers.

1 Introduction and Background

Many prior works [16, 13, 5–7, 15, 21, 10] build zk-SNARK proof systems for editing images or videos. This is a well-motivated and interesting use case for zk-SNARKs. There is a relevant, currently under development, standard called the C2PA [3]. The C2PA aims to add cryptographic keys to cameras which can sign generated images/videos to authenticate them as coming from a certain camera, and authenticate other metadata, like location or time of capture. These signatures are only on the original captured image/video. To preserve content provenance while editing the captured media, the C2PA organization suggests using trusted editing software. This leads to undesirable trust assumptions. Editing C2PA-signed videos is a good use case for zk-SNARKs/zero-knowledge proofs, however. One can generate a zk-SNARK showing that the media seen by a user is the result of applying an editing operation to an original signed video. This approach adds no other trust assumptions to such a system.

This could be used for many interesting applications. For instance, a journalist could generate a zk-SNARK to show that an image with a face blurred out was captured by a camera at a certain location/time. The zk-SNARK would hide the face of a sensitive subject, while still allowing readers to verify the image's authenticity. VerITAS's [5] introduction most directly discusses these use cases.

A *zkVM*, or zero-knowledge Virtual Machine, is a type of programming environment for zk-SNARKs which has been popular in recent years. Traditional zk-SNARK libraries, like Circom [12] or arkworks [4], require programmers to explicitly specify an arithmetic circuit representing the computation they wish to prove. This is often cumbersome, bug-prone, and difficult for programmers without cryptographic knowledge to learn. On the other hand, zkVMs build arithmetic circuits which can execute CPU instructions (often RISC-V instructions). The circuit is then used to generate zk-SNARKs showing that a certain program was

executed. Because the zk-SNARK circuit represents a CPU, programmers can write code in a traditional high-level programming language like Rust, and the SNARK circuit can prove the execution of assembly compiled from this code. zkVMs trade efficiency (loosely a 10-100× performance overhead) for ease of development/use, since using them is very similar to traditional software development. It is also easier to audit the computations proved by zkVMs, since their code is often written in a traditional high-level language, and can be audited using traditional software security techniques.

This note aims to record some lessons my collaborators and I learned from the failures we experienced while working on our paper "zk-Cinema: Proving Video Provenance in Zero Knowledge" [9]. First, I will discuss some difficulties around using zkVMs for the implementation components of papers on zk-SNARKs. Then I will discuss some difficulties around using Noir/Barretenberg for paper implementations, and give some thoughts on how to choose libraries for implementing zk-SNARKs for research papers. Finally, I will talk about some difficulties in making fair comparisons among prior works on image/video editing in zk-SNARKs, many of which may not be immediately obvious.

2 zkVMs (Failed direction 1)

Reasons for choosing a zkVM. The first research direction for this project was to implement common video editing algorithms inside of a zkVM. I will begin by summarizing the reasons why the PIs on our project initially chose to use a zkVM. Listing some of the reasons why some applied cryptographers in mid-2024 wanted to work with zkVMs:

- **Ease of training/developer experience**: Many professors working in applied zk-SNARK research at universities face a similar problem: they have a giant pool of undergrads and master's students (and junior PhD students) who are interested in working with them. However, common zk-SNARK libraries, like arkworks [4], bellman [2] or Circom [12] have steep learning curves. It is very difficult to utilize these students, since learning these libraries often takes up most of the time that a junior researcher is available to work on a project. zkVMs allow programmers to specify SNARK computations via Rust code, which largely solves this problem.
- **Twitter/Research hype**: Multiple zkVM companies had recently done funding rounds and/or were teasing token launches. This translated into a lot of hype on Twitter/X about these companies. A lot of this was from random Twitter anons, but there was some hype from serious researchers saying that the overhead of zkVMs wasn't too bad and that this was the future of zk-SNARKs. This led to a lot of genuine curiosity from our PIs about what zkVMs could be used for.
- **Large company support**: Common SNARK programming environments, like arkworks [4] or CirC [17] are academic projects, and don't have full-time maintainers. This makes working in these libraries difficult, especially for junior researchers. On the other hand, most zkVMs are backed by companies. Multiple zkVM projects that we worked with have Telegram/Discord chats where engineers from the company assist users with problems. The projects were also very actively maintained, with nearly daily commits to their GitHub repositories. This also led to an expectation that these systems would be more well optimized than comparable academic projects.

I will call these reasons the "Siren Song of zkVMs", since my advisor's research group started multiple zkVM-based projects based on this reasoning around mid-to-late 2024. I am also aware of other prominent zk-SNARK researchers who chose to use zkVMs for implementing experiments in their papers for similar reasons around this time, as more evidence that this argument was convincing.

Our initial system. Returning to our video editing project, we implemented our experiments using the Succinct SP1 zkVM [20]. Based on our explorations, this was one of the more mature zkVMs in late 2024, though there were a few comparable projects. Over the course of around 9 months, we built up a few implementations of video editing operations in a zkVM and made various optimizations. Initially, the optimizations were mostly around learning to properly write high-performance code for SP1. Later on, we worked with the

precompile system of SP1. Precompiles/inlines are a common performance tuning lever for zkVMs. For common operations, like hashing or signature verification, zkVMs offer specialized circuits for these operations, rather than executing the computations within the CPU abstraction. The performance of operations with precompiles is then comparable to that of proving them directly in a SNARK circuit. We implemented a custom precompile to speed up our video editing operations (specifically a precompile to do inner products over a finite field more efficiently).

We then submitted a paper based on this zkVM implementation. It was early-rejected. We encountered many difficulties in working with a zkVM in the research process, both before and after submission. The rest of this section summarizes the types of issues we encountered.

Difficulties with the research/review process: While many papers on the applications of zk-SNARKs can be summarized as "What if you implemented computation X in a zk-SNARK?", implementing some computation in a zk-SNARK does not make for good research on its own. To paraphrase my advisor [14], zk-SNARKs, by definition, can prove arbitrary computations and such a result wouldn't be very interesting. Good papers on applications of zk-SNARKs generally achieve novelty through technical contributions like interesting proof system tweaks, optimizations to their zk-SNARK circuits, or changes to the designs of systems/protocols using zk-SNARKs. As we learned from this project, many of these types of technical contributions are harder to implement when using a zkVM.

For instance, zkVMs are generally large, tightly integrated and optimized codebases. Consequently, they are somewhat hard to make proof system-level modifications to, especially for more junior researchers.

Furthermore, many of the traditional "tools of the trade" for optimizing zk-SNARK circuits are not available or much harder to use in a zkVM context. When we were working on this project, fast addition/multiplication operations in the zkVM's "native field", "free" linear combinations, fast SNARK-friendly hashing (e.g. Poseidon), pseudorandom challenges derived from the proof's witness, ways to take advantage of "small" values and ways to take advantage of repeated/parallel structure of computations like folding were not supported or much more expensive in the SP1 zkVM than in a regular zk-SNARK context. Furthermore, SP1 did not have a particularly ergonomic system for generating precomputed advice for SNARK computations. These tricks represent a large fraction of the common tricks used in zk-SNARK research to optimize circuits in interesting ways. The lack of these features actually led to a lot of friction in communication, since the professors in our group expected that we would have access to these tricks, and found the constraints imposed by a zkVM to be unnatural.

We were able to come up with some optimizations to our zkVM code (in some cases making the code multiple times faster), but most of the optimizations were very specific to SP1, with some of these documented in [8]. One obvious optimization was using the previously discussed *precompile* system. Precompiles are also often not very technically novel, and I discuss more issues with them below.

In addition to zkVMs making it harder to make novel technical contributions, we also found that they led to difficulties in the peer review process. Most academic research projects are conducted with the goal of submitting a paper to a conference. Reviewers for applied cryptography tracks at security conferences are (generally) educated about their field, and are aware that zkVMs are easy to use, and often just involve writing Rust code. This often prompts reviewer comments about a paper not doing very much engineering work, or the paper not doing any proof system-level work. Empirically, every paper which uses a zkVM or another "beginner-friendly" library for its implementations that I have submitted to a security conference has gotten these types of comments. While these comments don't make paper acceptance impossible, they do mean that the paper is starting at a disadvantage.

Developer Experience: One of the most compelling points in favor of using a zkVM was the ease of development, which would make it easier for junior researchers to contribute to our project. While SP1 did make it easier for the other junior students and me to contribute to the project, the benefit was perhaps less than expected. First of all, it turned out that the difficulty of using arkworks/other programming environments is a case of "cascading bottlenecks" or Amdahl's law. While other SNARK environments are difficult to program in, the junior researchers we were working with (myself included) were still new to cryptography, and there were still other obstacles to them contributing to our project. The difficulty of programming zk-SNARKs was just the most noticeable bottleneck in getting students ramped up on zk-

SNARK research (less visible bottlenecks include research/system design taste, understanding the overall system/security model, proposing optimizations, etc.), and removing the obstacle of coding had more modest productivity benefits than expected.

As a side note, a similar effect likely applies to the use of AI coding agents in applied cryptography research. While they make programming much easier, they just move the bottlenecks in research productivity to things like research taste or verifying the quality of AI outputs,[1] which tend to be difficult to do for new researchers. Furthermore, AI coding agents trivialize most of the low-level implementation work that junior researchers are typically assigned in applied cryptography projects. AI coding will likely cause major changes to how junior researchers are typically utilized on applied cryptography projects.

To discuss some smaller development issues, the developer experience for implementing precompiles in SP1 was difficult at the time, and was a lot harder than specifying circuits in arkworks/Circom. This led to a developer learning curve where one of our master's students had to go from writing basic Rust code to writing fairly involved AIR constraints in Plonky3. Secondly, at the time, every zkVM that we worked with had very long compilation/execution times, while environments like arkworks had very fast build-test-fix loops. There were some other annoying developer ergonomics issues with SP1 at the time, like a sometimes painful version upgrade/management process. These smaller points all made SP1 less developer-friendly than one might initially expect.

Performance: Lastly, and most straightforwardly, zkVMs tend to have major performance overheads. This is especially relevant for the specific problem of video editing in zk-SNARKs. Even if a circuit just performs a few constraints of work per pixel of an edited image/video, realistically-sized images/videos have millions of pixels, and these will stress the performance limits of modern zk-SNARK systems. Consequently, the 10-100× overhead of a zkVM is particularly painful for this problem domain.

One way we tried to deal with this performance overhead was by implementing both baseline and optimized versions of our algorithms in a zkVM to control for zkVM overhead. While this benchmarking technique has value, there are still potential issues with the approach. In particular, not every computation has the same overhead when proved in a zkVM. For instance, computations with precompiles, or computations on integers have similar costs in a zkVM to proving them in a traditional zk-SNARK. On the other hand, operations like hashing, taking input/output/precomputed advice or finite field operations were more expensive in SP1 at the time than they would be in a zk-SNARK [8]. This cost distortion meant that an improvement to zkVM code may not transfer cleanly to an improved circuit for zk-SNARKs.

Concretely, our system achieved a performance of about 620 microseconds per edited pixel of content, while the concurrent state of the art, Eva [21], achieves 2.6 microseconds per pixel, or a throughput difference of about 240×. To give an idea of the contributors to this performance gap, about 3× of this gap could be attributed to the state of the art work using faster hardware than ours. Around another 5× could be attributed to using suboptimal algorithms, and there was a base overhead of 10-20× in the underlying proof system. These are loose estimates based on my research notes, since it was difficult to rerun benchmarks with the version of SP1 that we used for the initial submission, so these shouldn't be taken overly seriously. It could be argued that this means that the "true" zkVM performance overhead was thus 10-20×, but this misses some subtlety. Some of the approximate 5× algorithmic overhead was due to SP1 not supporting the algorithmic/implementation tricks used by faster systems. While this may not be a performance overhead in a purely cryptographic sense, it is an overhead to an implementor/user of the zkVM.

While we tried to argue that our zkVM implementation was just showing the utility of various algorithms proposed in the paper, it is kind of hard to overcome the issue of being multiple orders of magnitude slower than prior works in the area. As previously mentioned, this paper was rejected. We did not see a way forward with a zkVM-based system due to the previously discussed reasons, and we reimplemented our system entirely.

[1] This includes verifying that AI-generated code is secure!

3 Noir (Failed direction 2)

After our first submission was rejected, we rebuilt our video editing system using the Noir/Barretenberg programming environment [1] to implement zk-SNARK circuits, and put together a second submission. Noir/Barretenberg is another beginner-friendly SNARK programming environment. It lets programmers write SNARK circuits in a programming language which looks like Rust. These programs are then compiled into circuits and proved using a backend called Barretenberg (though other backends exist). Noir/Barretenberg is generally faster than a zkVM, it gives programmers relatively tight control over the arithmetic circuits generated, and it allows programmers to use many common optimization techniques, including lookup arguments [11] or zk-RAM [18], very easily, which can otherwise be hard to use.

The system we built for our second submission was "only" about 10× slower than the prior state of the art. This submission was also early-rejected. Using Noir/Barretenberg in a research context has some of the same issues as using zkVMs for research, though to a lesser extent, because Noir/Barretenberg is faster and gives more control to users. We did encounter some performance/developer experience issues specific to Noir/Barretenberg which I think are worth documenting:

- **Compilation time/resources**: The Noir compiler's resource usage/compilation time is a well-known difficulty of working in the Noir ecosystem. For circuits with more than a few million gates, the compiler uses tens of gigabytes of RAM, and can take multiple hours to finish compilation. This is because the compiler makes some very heavyweight compilation passes, and gives very few options for programmers to configure this compilation process.
- **Witness generation time/witness handling**: In our benchmarks, witness generation for circuits took 20-50% as long as actual proving. To contrast, in libraries like arkworks, witness generation is 1-10% of proving time. While some papers don't count witness generation towards their performance numbers, this is part of a prover's workflow, especially in our system where witness generation couldn't be done all at once because of memory limitations and needed to be interleaved with proving for our large statements. Secondly, Noir handles witness generation in a separate command from proving. To pass around witness data/proof inputs, Noir passes these around as TOML files. This was not very ergonomic from a programming perspective, and there were added overheads from having to repeatedly parse/write large TOML files throughout the proving pipeline.
- **Other performance irregularities**: Noir has some other performance irregularities which we were not able to fully understand while working on the paper. For instance, looking at the logs outputted by the Barretenberg `bb` prover, about 50% of the prover time is spent before a log message that says "prover key generated". We were not able to find documentation for what this log message means. Assuming a prover key was being generated, it was not possible to cache this operation. Also, though this wasn't relevant for this iteration of the paper, Barretenberg is not very efficient for small circuits, and takes 500 ms for even tiny circuits because it has certain fixed work which must run for circuits of any size [19].

The Noir-based version of our paper was early-rejected, again partially because the approximately 10× performance gap was hard to defend. To give an idea of the contributors to this performance gap, expensive witness generation, costs of moving around witnesses, and repeatedly re-parsing the trusted setup and re-setting up the prover were reasonably large contributors (maybe a 2× contributor, though this is hard to precisely measure). This leaves a core overhead of 5×, though again, this is a coarse estimate. It is possible that this could have been accepted somewhere with better framing. However, for our third submission, we switched to a custom folding-based prover, which is the current state of this project. We got our performance numbers down to 3.4μs per pixel. The reasons for this improvement were better hardware utilization/parallelism, faster witness generation, a variety of other proof system optimizations [9, Appendix J], and other optimizations in the `vega-prover` repository.

4 Lessons around culture/research practices

My collaborators and I spent a long time working on the research directions in the previous two sections. One may ask why we spent so long on these directions without seeing something was wrong. I think there are a few lessons around research culture/practices to be derived from this experience.

First of all, we never really did a back-of-the-envelope estimate for the expected overall performance of our system. We didn't do this during the first few months of our project because it took a few months to be confident that we were writing code for SP1 in a reasonably optimal way. Secondly, we did not have a specific performance goal in mind, and were primarily aiming to showcase the impact of some algorithmic optimizations. The relevant prior work Eva was not publicly available when we started this project, so we did not really have a performance goal to hit. One lesson that I have drawn from this experience is that "not embarrassingly slow" is a valuable performance target to aim for in this type of applied cryptography research, even if there isn't an explicit comparison point to aim for.

Furthermore, there were some other broader issues with our research culture. At a few points during the project, this was not really anybody's top priority, and we were all pretty apathetic about it. It sometimes felt like there was pressure to get submissions of papers pushed out. The bulk of the concrete work on this project was done by junior researchers, and we sometimes didn't raise issues because of not knowing when to/feeling intimidated. These all contributed to a culture where we didn't really question our assumptions/methods. These are common failure modes, but this has taught me that these "vibes" level things are really important on a research project.

5 Considerations in choosing a zk-SNARK programming environment

Sections 2 and 3 lead to the logical question of how one *should* pick a zk-SNARK library for research implementations. This is a complicated question, and I don't think there is currently a single "safe default" option, though I think this complexity is what makes research in applications of zk-SNARKs interesting. In general, choosing a zk-SNARK programming environment is a tradeoff between ease of use and speed/ability to make novel optimizations, though specific systems may be more well-suited to specific tasks. I think researchers should look at what SNARK optimization levers they expect to need (the list in Section 2 is a decent collection), as well as how competitive their specific research area is in terms of system performance, and use this to choose the easiest to use SNARK environment which meets these goals. In general, I think that AI coding assistants (**used responsibly**) will (and should) raise the quality bar for paper implementations in applied cryptography, and that this should be factored into implementation library choices.

Continuing the discussion of choice of programming environment, papers which use zkVMs for their experiments continue to be released, in spite of my arguments in this note. This leads to the question of what zkVMs could be useful for in research. Below are some qualities of research problems which I think are well-suited for zkVM-based system implementations:

1. **Little/no interest in SNARK performance/internals**: Some research works really do not require good performance. There may not be related prior work, the work is just trying to show the feasibility of an approach or some other aspect of the research is very interesting.
2. **Hardware availability**: Many zkVMs have nice support for GPU-accelerated proving and distributed proving across multiple workers. It is often easy to compensate for the overhead of zkVMs by using much more powerful hardware.
3. **Ease of coding is important**: In some systems, it is important that users can easily specify computations proved with zk-SNARKs, like in a system for secure smart contracts.

I think most research papers which use zkVMs for implementation hit one or multiple of these points.

6 Difficulties around fair benchmark comparisons

Lastly, we missed some simplifying assumptions/drawbacks of prior papers which made fair comparisons between our works difficult. The process we used to perform literature reviews for this work involved asking a student to read some paper, and then report back with a few minutes of discussion about the relevant paper, as well as writing a few sentences in our paper's Related Work section. This approach was problematic because some of these simplifying assumptions/drawbacks are not obvious in prior works, and we did not even know what to look for at first because we had not spent time running/optimizing experiments. As a general disclaimer, I think these prior works all make valuable contributions to the research literature, and all the listed simplifications are acceptable on their own. However, in aggregate they make for a difficult-to-navigate literature. Below, I list a few categories of issues which make honest comparisons between prior works on zk-SNARKs for image/video editing difficult:

Different color space representations: Digital images/videos are generally represented as arrays of pixels. Pixels are blocks of color, which can be represented in different "color spaces". The most well-known pixel color format is "RGB", which assigns 3 values between 0 and 255 to represent a pixel's color in terms of intensities of red, green, and blue. Grayscale, or black and white images, only require one byte per pixel to represent their brightness. Another common color space is YCbCr, where the Y component represents brightness, and Cb/Cr represent "chroma blue"/"chroma red" respectively [21]. Videos typically perform "chroma subsampling," where each pixel receives one "Y" value, but each block of 2×2 pixels only receives one Cb/Cr value. This has the effect of representing an uncompressed video/image in 1.5 bytes per pixel rather than 3.

Generally speaking, working with RGB images in SNARK circuits is about 3× more expensive than working with grayscale images, and about 2× more expensive than working with subsampled YCbCr images, approximately proportional to the amount of data required to represent the image. Most prior works in this area use the RGB color space. VerITAS implements some of their editing operations only for grayscale images [10, Page 6],[5]. Eva works in the YCbCr color space with chroma subsampling [21, Section 3.1]. As a result, components of these papers are faster by 2-3× because of a choice of pixel representation.

Specific choices of video editing operations: There are many different photo/video editing operations, and many prior works choose particularly easy ones to implement. A common editing operation implemented in prior works is grayscaling, or converting a color image to black and white. Grayscaling outputs a 1-channel image, meaning that handling an editing circuit's output is approximately 3× cheaper than for a color output, as discussed in the previous paragraph. HyperVerITAS only implements editing operations which decrease the output's size, which saves output handling costs: cropping to decrease an image's size by 50% [10, Table 4] and grayscaling.

Grayscaling satisfies another property which makes it particularly easy to work with, namely that grayscaling operates on each pixel of an image/video *independently*. Eva only implements edits where each pixel is edited independently.[2] This includes edits like brightening a pixel, or inverting the colors of a video. Operations like blurring a video or resizing a video do not meet this requirement, since the values of output pixels depend on the values of pixels around them. This is a somewhat limited set of operations. The reason for this choice of operations is that Eva splits input media into blocks of pixels, and edits each block of pixels independently. The prior works VIMz [7] and TilesProof [6] also use this approach. Implementing edits like blurring or resizing requires authenticating pixels from adjacent blocks to the one being edited, which increases costs of editing operations. The Eva paper proposes using additional Merkle inclusion proofs to authenticate adjacent blocks of pixels, but these would likely substantially increase circuit costs [21, Table 3].

Verification times/proof sizes: The S in zk-SNARK stands for succinct, meaning that proofs should be short and fast to verify. zk-SNARKs for image/video editing have large public outputs corresponding to the final edited media. This means that in practice, many works in this area have quite long verification times,

[2] Their prover can prove edits on *macroblocks*, or 16×16 blocks of pixels, though they only actually implement per-pixel edits.

though this is for an interesting and diverse collection of reasons. In general, the underlying zk-SNARK systems used by these works do have cheap verification and short proofs, but the way they are used leads to issues with succinctness.

As previously mentioned, the main issue at hand is with *large public outputs*. The zk-SNARKs in these works have an entire image/video as a public output of the circuit, and the proof must somehow cryptographically bind the output to the zk-SNARK proof. A common approach is hashing the output video with a SNARK-friendly hash function, as in Eva and VIMz.[3] In Eva, the cost of applying this SNARK-friendly hash to a 2-minute output video is 232 seconds (on a 16-core CPU), nearly twice the time to watch the video.

There are two other categories of associated issues. TilesProof [6] splits images into blocks, and generates a separate Groth16 proof for each block. In TilesProof, the final edited media is a public output of this collection of Groth16 proofs. Verifying the public outputs of these Groth16 proofs involves computing multiscalar multiplications of size equal to that of the final output media. TilesProof achieves verification times of 3 seconds for a single 720×1280 image.

VerITAS and HyperVerITAS build specialized approaches to verifying matrix multiplications. These lead to relatively long verification times, because part of the verifier's work involves an $O(n)$ time matrix multiplication [5, 10]. Concretely, for an image of 2^{25} pixels and the grayscale edit, HyperVerITAS has verification times of 100 seconds [10, Table 9]. VerITAS has verification times of 15 seconds for images of $\sim 2^{25}$ pixels, though they have an approach for decreasing the verifier's time at the cost of a $>2\times$ increase in prover time [5, Section 7.1]. As previously explained, these verification times are more or less linear in the size of media verified.

Miscellaneous: Related to the prior discussion of implementations of video editing operations, some prior works have other interesting simplifying assumptions. For instance, Eva uses the Griffin SNARK-friendly hash function which is approximately $2\times$ cheaper in circuit than the standard Poseidon [21, Section 6]. HyperVerITAS builds custom sumchecks for certain categories of editing operations, rather than arbitrary circuits as prior works do.

7 Conclusion

To conclude, this note records many of the learnings my collaborators and I had while working on our paper, zk-Cinema [9]. The goal of this note is to formalize various "folklore" knowledge that I learned throughout this project. I discuss some drawbacks of the beginner-friendly zk-SNARK libraries we tried, some of which may not be immediately obvious. I also catalog some of the interesting quirks around benchmarking prior works on image/video editing in zk-SNARKs. I hope these are useful for future research on zk-SNARKs for video/image editing, as well as more general work on applications of zk-SNARKs.

Acknowledgements: I would like to thank my coauthors Dan Boneh, Ian Miers, Trisha Datta, Jianfeng Guo and Xinyi Zhao for their help with the research this extended abstract is based on. I would also like to thank the ARCS Foundation and DARPA for their support.

Disclosure: At the time of writing, Alexander Frolov was an intern at a16z crypto research, which builds a competing zkVM to Succinct SP1 called Jolt. To see Andreessen Horowitz's standard disclosures, please visit https://a16z.com/disclosures/.

References

1. Aztec Labs: Noir: A domain specific language for SNARK proving systems. https://github.com/noir-lang/noir (2022), accessed: 2026-06-21

[3] VIMz doesn't benchmark this step, though it is required for verification.

2. Bowe, S., Grigg, J.: bellman: zk-SNARK library. https://github.com/zkcrypto/bellman (2023), accessed: 2026-06-21
3. Coalition for Content Provenance and Authenticity (C2PA): C2PA specifications, version 2.4. Technical specification, Coalition for Content Provenance and Authenticity (2026), a Joint Development Foundation project. Accessed: 2026-06-21
4. arkworks contributors: `arkworks` zksnark ecosystem (2022), https://arkworks.rs
5. Datta, T., Chen, B., Boneh, D.: VerITAS: Verifying image transformations at scale. IEEE S&P 2024 (2024), https://eprint.iacr.org/2024/1066
6. Della Monica, P., Visconti, I., Vitaletti, A., Zecchini, M.: Trust Nobody: Privacy-Preserving Proofs for Edited Photos with Your Laptop. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 4624–4642. IEEE Computer Society, Los Alamitos, CA, USA (May 2025). https://doi.org/10.1109/SP61157.2025.00014, https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00014
7. Dziembowski, S., Ebrahimi, S., Hassanizadeh, P.: Vimz: Private proofs of image manipulation using folding-based zksnarks. PETS 2025 (2025), https://petsymposium.org/popets/2025/popets-2025-0065.pdf
8. Frolov, A.: Trade-offs and pitfalls in zkVM design (or, some ways to make your zkVM code 3–10 times slower). Blog post (Mar 2025), https://www.sasha.place/blog/2025/03/zkvm-design-tradeoffs
9. Frolov, A., Guo, J., Zhao, X., Datta, T., Boneh, D., Miers, I.: zk-cinema: Proving video provenance in zero knowledge. Cryptology ePrint Archive, Paper 2026/1598 (2026), https://eprint.iacr.org/2026/1598
10. Greiner, G., Mowery, T., Soni, P.: HyperVerITAS: Verifying image transformations at scale on boolean hypercubes. PETS 2026 (2026), https://eprint.iacr.org/2026/641
11. Haböck, U.: Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530 (2022), https://eprint.iacr.org/2022/1530
12. iden3: circom: zksnark circuit compiler. https://github.com/iden3/circom (2025), accessed: 2026-06-21
13. Kang, D., Hashimoto, T., Stoica, I., Sun, Y.: ZK-IMG: Attested images via zero-knowledge proofs to fight disinformation. arXiv 2211.04775 (2022)
14. Kaptchuk, G.: Paper pitfalls: Lessons from reviewing. Blog post (Aug 2023), https://www.cs.umd.edu/ kaptchuk/blog/post/reviewing-reflections.html
15. Ko, H., Lee, I., Lee, S., Kim, J., Oh, H.: Efficient verifiable image redacting based on zk-snarks. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. p. 213–226. ASIA CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3433210.3453110, https://doi.org/10.1145/3433210.3453110
16. Naveh, A., Tromer, E.: Photoproof: Cryptographic image authentication for any set of permissible transformations. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 255–271 (2016). https://doi.org/10.1109/SP.2016.23
17. Ozdemir, A., Brown, F., Wahby, R.S.: CirC: Compiler infrastructure for proof systems, software verification, and more. IEEE S&P 2022 (2020), https://eprint.iacr.org/2020/1586
18. Setty, S., Thaler, J.: Twist and shout: Faster memory checking arguments via one-hot addressing and increments. Cryptology ePrint Archive, Paper 2025/105 (2025), https://eprint.iacr.org/2025/105
19. Shih, M.: Personal communication. Personal communication related to another project (May 2026)
20. Succinct Labs: SP1: A zero-knowledge virtual machine. https://github.com/succinctlabs/sp1 (2024), accessed: 2026-06-21
21. Zhang, C., Yang, X., Oswald, D., Ryan, M., Jovanovic, P.: Eva: Efficient privacy-preserving proof of authenticity for lossily encoded videos. IEEE S&P 2025 (2024), https://eprint.iacr.org/2024/1436