SoK: Compiling programs for integration with multiple ZKP systems

Guillaume Drevon

gd@sikoba.com

2 February 2020

Abstract

Generating a zero-knowledge proof of the execution of a software program is a task that has been solved in theory many years ago, but in practice many hurdles remain to be overcome. In this paper, we explore the results of our work trying to solve this problem in the most generic way, using the *isekai* verifiable computation framework. In addition to the issues we have faced, we also show some optimizations that we found along the way and demonstrate how we used our project to benchmark several zero-knowledge proofs systems.

This document was written in response to the call for papers of the 3rd ZKProof Workshop, under the Systematization-of-Knowledge track.

1 Background and Motivation

Zero-knowledge proofs is a very active field of research. New proof systems are being discovered regularly, with each one having unique properties that make it relevant for some use cases. We can for instance cite:

- recently proposed schemes based on Polynomial Commitments such as Marlin [CHM$^+$19] or Plonk [GWC19], which have a universal trusted setup;
- Bulletproofs [BBB$^+$17] which is based on the discrete logarithm problem and require no trusted setup;
- schemes such as ZK-STARKS [BSBHR18] and Fractal [COS19], which are based on Interactive Oracle Proofs and provide post-quantum resistance;
- and ZK-SNARKs [Gro16], which have constant proof size.

However the variety of the systems is also a curse because it is difficult to evaluate and compare all of them.

Although zero-knowledge proofs (ZKP) are defined over any formal language, most ZKP implementations are specific to a particular zero-knowledge scheme, usually implemented by the team who wrote the scientific paper, and based on low-level ZKP friendly language such as QAP, R1CS or circuit. Other implementations are usually wrappers over one zero-knowledge scheme implementation and are usually based on a domain-specific or even a low-level language. Examples of such projects are ZoKrates [ET18], c0c0 [KZM$^+$15], Circom[1] and snarky[2].

There are a few implementations that do consider high-level languages, but they are all dedicated to C and ZK-SNARKs. Also, these implementations are more than five years old and do not seems to be under active development. We can cite the seminal paper of Pinocchio [PGHR13] and the improvements made in Geppetto [CFH$^+$14], Pantry [BFR$^+$13], which bring dynamic memory but at a high cost, TinyRAM [BSCG$^+$13] which significantly improves dynamic memory performance though a permutation network and Buffet/pequin [WSH$^+$14] which removes some of TinyRAM's overhead.

In light of those observations we have decided to work on a middleware that would consume computer programs written in popular high level programming languages and be able to integrate with existing ZKP systems. Our work is based on Pinocchio and extends it in several ways. Our approach is unique in the sense that we do not know of any other project that supports natively several high-level programming languages and several zero-knowledge schemes. This paper is a journey into the making of isekai[3], an open-source verifiable computation middleware developed by Sikoba Research.

2 Introduction

isekai is a verifiable computation framework whose aim is to allow users to work with their preferred programming language, while being able to choose between multiple verifiable computation systems. Thus they can test multiple options without having to make difficult choices at the beginning of a project.

The central ingredient of isekai is R1CS as defined in [BCG$^+$13]. isekai transforms a program into a system of constraints that are satisfied by the execution trace of the program. ZKP systems can then apply cryptographic functions to this set of equations. R1CS is a low-level language that can represent any arbitrary arithmetic or Boolean circuit and can be reformulated for use in specific ZKP systems. We refer the reader to the previous ZKProof workshop [4] to see why it has become a standard.

In practice, isekai converts the input program into LLVM Bitcode[5], which is in turn transformed into a circuit representation. The main components of isekai are:

1. The **frontend**, which parses source code and transforms it into an intermediate form;
2. The **backend**, which takes the intermediate form and produces either an arithmetic or Boolean circuit;
3. The **reducer**, which takes the arithmetic (or Boolean) circuit and produces R1CS with its assignments;

[1]https://github.com/iden3/circom
[2]https://github.com/o1-labs/snarky
[3]https://github.com/sikoba/isekai
[4]https://zkproof.org/events/#0ea671e4-da62-7
[5]https://llvm.org/docs/LangRef.html

4. The **prover**, which takes R1CS and generates a proof of execution.

This modular architecture allows for a simple workflow that follows a waterfall pattern (see figure 1). It has been designed to facilitate the support of additional programming languages, the integration of several zero-knowledge proof libraries and the implementation of common high-level programming language features.

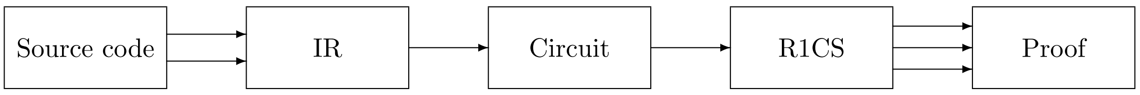


Figure 1: isekai workflow

In this paper we explain several techniques that we have used for integrating generic computer programs and several ZKP systems. The main contributions of the paper are the following:

- Handling of LLVM Control flow
- Optimisations around dynamic memory access
- Benchmarking of ZKP systems

The remaining part of the paper is structured as follow. In Section 3 we expose the issues we faced when parsing LLVM bitcode, in section 4 we explain some non-determinism optimisations we found and in section 5 we present benchmarks of several ZKP system, as measured with isekai.

3 LLVM Frontend

Many programming languages exist and are in use nowadays, each one having various unique features. To name a few, we can think of pointers and pre-processor for C, STL in C++, OOP for Java, the memory model of Rust, functional programming with Haskell, and so on.

Trying to support several languages natively would obviously be a never ending task. Instead, we want to benefit from compilers, and in particular their internal representation of source code, which is both language and hardware independent. This is why we decided to use LLVM which provides a compiler toolchain targeting arbitrary programming languages and exposing compiler modules.

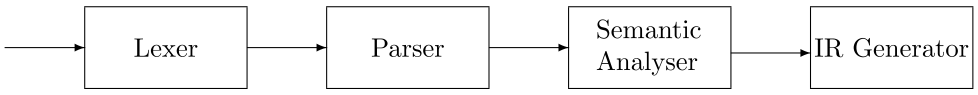


Figure 2: Pipeline of a Compiler's Frontend

3.1 LLVM Bitcode

LLVM bitcode aims to be a "universal IR" of sorts, by being at a low enough level such that high-level ideas may be cleanly mapped to it, while being independent of any hardware component. One particularly attractive feature of LLVM bitcode is the Static Single Assignment (SSA) representation. This means that each variable in LLVM is assigned exactly once. This property is useful for many compiler optimisations and at first glance it seems well adapted to R1CS representation, although there are many subtleties to take into account.

Integer type

We only support the integer type of LLVM, which can have an arbitrary (but known at compile time) bit width. Arithmetic operations must then be done modulo the bit width, which is not compatible with the modulo-p arithmetic of the R1CS. As a result there is an induced cost in term of overflow checks.

Programming Language

isekai is currently able to parse LLVM bitcode which is generated from C or C++ source code. It should be easy to adapt isekai to work with bitcode compiled from other languages. Languages with LLVM compliers include C, C++, C#, Crystal, D, Haskell, Julia, Kotlin, Lua, Objective-C and Rust.

3.2 Control flow

Even if a variable is assigned at only one instruction thanks to SSA representation, this instruction is not necessarily called only once, as blocks (of instructions) can be re-entered. This is typically the case with loops. On the other hand, the arithmetic circuits that we want to generated are DAG (directed acyclic graph) whose vertices are operators having inputs represented by its incoming edges and outputs by the outgoing edges. Operators, as their name suggests, are doing arithmetic operations on integers (e.g addition and multiplication). Boolean circuits are defined in a similar way using boolean operations.

As you can see there is no form of loop or jump. Loops must then be represented by duplicating the code, constrained by a maximum number of iterations when the bound is not known at compile time. For performance reasons, the maximum can be provided by the user before any loop.

However, LLVM IR can still be any arbitrary control flow graph (CFG). It could be in theory realized as a sequential code with "if" and "while" statements, according to the Böhm-Jacopini theorem [BJ66], but the theorem does not come with an efficient algorithm.

We found the following projects that are tackling this problem of compiling LLVM IR to languages using high-level control flow operators only (no goto):

- Emscripten, the LLVM IR-to-JavaScript compiler uses the "Relooper" algorithm [Zak11]. This algorithm is quite complex, and also introduces a significant performance penalty.
- LLVM WebAssembly backend with its "Stackifier" algorithm [6]. Of comparable complexity, it also introduces the same performance penalty in the worst case (if the CFG is "irreducible"). Moreover, it requires multi-level break statements, which we would have to emulate with additional variables.

On top of that, since LLVM does not have loop instruction such as *for* or *while*, we still need to recognize a loop. It turns out this is a very difficult questions as compilers can be very creative with their optimisation techniques. Over the last decade, the compilers have started to implement a number of aggressive optimizations such as:

[6]https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2

- split a loop (loop fission, loop splitting, loop un-switching)
- merge loops (loop fusion)
- swap inner and outer loops (loop interchange)
- rewrite nested loops (loop skewing, loop nest optimization)
- etc...

3.3 Decompilation algorithm

Because of all these problems, our solution supports only control-flow graphs that can be obtained by compiling a C99 code without *goto*, *break*, *continue*, and *return* statements, using no aggressive optimisation. In this case, control flow instructions such as *if, for, while* and *do* would usually fall into two possible patterns having a basic block ending with a conditional instruction. We call them loop pattern and conditional pattern.

In order to identify in which pattern we are, we look at the least common ancestor of the conditional instruction (in the inverted shortest path tree, starting from the next LLVM return statement) which represent the end of the loop or condition. Then by looking at the children of the conditional instruction, we can say which pattern it is. Note that the least common ancestor can be found efficiently using Tarjan's off-line lowest common ancestor algorithm [GT83], the complexity being linear in the number of edges and vertices.

In fact because of short-circuit evaluation semantics (for instance depending on left expression in *AND* and *OR* operators), more complex patterns can appear, but the general idea is to check whether we match one of the two patterns and if not, we perform code duplication.

3.4 Testing

In order to validate our construct, we have implemented a test engine that is able to execute an arithmetic circuit. It does the following:

1. Look at test cases defined in the test folder
2. Compile source code and execute the compiled executable using the input files
3. Use isekai to generate the arithmetic circuit
4. Interpret the circuit using the same inputs
5. Check that the outputs are the same as with the program execution

This way we can ensure the circuit is a true representation of the program. We have currently 66 test cases checking many different C or C++ features. Moreover, a test case can have several inputs in order to validate different behaviors.

3.5 Memory access

In LLVM, variables can be changed via allocations on the stack as memory is not following SSA. This means we need to support pointers and dynamic array indices. When we try to access a memory location pointed by a non constant value, our first implementation was to check every possible memory address (for an array it would be all elements of the array). This solution is not efficient and induces a cost linear in the memory size every time the memory is accessed (for read or write), although we were able to improve a bit the performances as explained below in section 4.3.

4 Taking advantage of R1CS

4.1 Introduction

Since the constraints are living in a finite field, there are several ways to express the same constraint, and surprisingly this can slightly affect the performance of the zero-knowledge scheme. Similarly, eliminating linear constraints (which are not bi-linear), although it effectively reduce the number of constraints, can sometimes degrade performance. So one must carefully check whether an optimisation is really improving the performance.

4.2 Advanced arithmetic

Working at the constraints level can sometimes provide some shortcuts. This is the case for instance when one writes a division (in the native field) as a multiplicative constraint. Following this principle, we added divide gates for field division and also for integer division. This allowed us to support division and modulo operators in C and C++, for both signed and unsigned types.

4.3 Dynamic memory

In order to model memory access where both the memory and the pointer are variables depending on the inputs of the program, i.e they are not constant values, we were checking for all possible memory locations the pointer could address. This resulted into $3 * n$ constraints if n is the memory size.

We were able to improve over this first implementation by introducing a new gate that we call *arithmetic split.*

A-Split Gate
Given input $b \in \mathbb{F}_p$ and a non-null integer n s.t. $b < n < p$, the a-split$_n$ gate outputs n wires $b_0, ...b_{n-1}$ that are all zero, except $b_k = 1$ when $k = b$. This gate can be implemented using $n + 2$ constraints:

- n constraints saying that outputs are booleans (0 or 1)
- one constraint expressing that the outputs are all null except one ($\sum_i b_i = 1$)
- one constraint stating that the non-null element corresponds to b ($\sum_i i * b_i = b$).

Using this gate, we can effectively compute a[b], where a is an array of size n, using $2 * n + 2$ constraints, as shown in figure 3

There is still room for improvement as you can see below, but the following optimisations are not yet implemented into isekai.

The first point that one would notice is that we are using two constraints that are not bi-linear but only linear, so it is possible to combine them and reduce the overall number. With this small improvement we can get down to n constraints for the arithmetic split:

$$b_i * \sum_{k=0}^{i-1} b_k = 0, i = 1, .., n-1$$

$$\sum_i b_i = 1$$

A better optimisation is to use binary-search techniques. Binary search usually does not work well with constraint systems as we need to track both branches, so effectively doubling the halving...

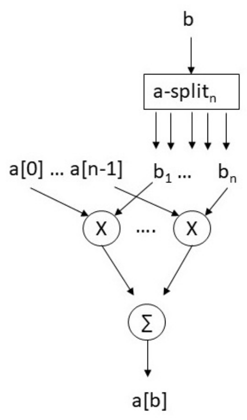


Figure 3: Using a-split gate for array look-up

However here the trick is that an array of size two can be accessed by only one constraint, so one constraint can halve an array in two arrays, giving in total $1 + 2 + 4 + ... + 2^q = 2^{q+1}$ constraints where q is the bit length of n. That means that instead of $2n$ constraints with the arithmetic split gate, we can achieve something between n and $2n$, depending on how close n is to a power of 2.

4.4 Constant array

When the input array is constant, the previous technique generates n constraints. However we can take advantage of the array being of constant values to reduce the cost even more, by using polynomials.

4.4.1 Evaluating olynomials

First let's see how we can evaluate a polynomial using bi-linear constraints system. A basic observation is that monomials can be trivially computed from their previous monomial: $x^k = x * x^{k-1}$, which means a polynomial can be computed by at most $n - 1$ constraints where n is the polynomial's degree.

The idea then is to lower the degree of the polynomial by doing Euclidian division. With the constraint $P - R = B * Q$, we can compute polynomial P using lower degree polynomials R, B and Q. If we keep repeating this process with the results of Euclidian division, we can constantly reduce the overall degree of the resulting polynomials. We observe here there is a race condition on the number of constraints, increasing with the number of divisions and polynomials, and decreasing with the degree of the polynomials. It turns out we can win this race, the minimum of constraints being obtained when the resulting degree is close to $\sqrt{n}$, giving approximately $2\sqrt{n}$ constraints.

4.4.2 Array look-up

Going back to our initial problem of doing an array look-up for a constant array a of size n, it can be done in $O(\sqrt{n})$ by using the polynomial of degree n-1 s.t. $P(i) = a[i]$ for all $i = 0...n - 1$

4.5 Native field arithmetic

Enabling ZKP for generic computations is a noble cause, however ZKP are often used in the context of cryptography context, which means doing arithmetic over a field of large prime order. If one wanted to do this with isekai, one would need to implement big integers in C or C++, as we only support integer types. Since everything at the end is getting translated into arithmetic over a big prime field, it does not sound right to convert big integers into integers, and then convert them back into big integers.

We have decided to avoid such overhead by having a custom type for which isekai will not perform any overflow checks. Using this type, called *Nagai*, one can model the arithmetic over the underlying field of the chosen zero-knowledge scheme. As an example, we have implemented basic elliptic curve operations in a C++ header so that doing a elliptic curve multiplication for any curve can be done as easily as this:

```
{
#include "nagai/ecc.hpp"

struct NzikInput {
    Nagai* a;
    Nagai* b;
    Nagai* c;
};

struct Output {
    Nagai* x;
};

extern "C" {
    void outsource(struct NzikInput *, struct Output *);
};

void outsource(struct NzikInput *input, struct Output *output)
{
    Field x = Field::copy_from(input->a);
    Field y = Field::copy_from(input->b);
    ECC  c(40962,1);
    ECCPoint g = c.NewPoint(x,y);

    output->x = (Nagai*)(g.Multiply(Field::copy_from(input->c)).x);
}

}
```

Listing 1: Elliptic curve point multiplication for isekai

This sample code, doing point multiplication for a given curve, generates 3,564 constraints. As a comparison, the handcrafted optimizations for a dedicated curve made by the talented Zcash team resulted in 3,252 constraints [HBHW19]

5 ZKP Benchmarks

This results presented in this section are taken from [DK19].

5.1 Integrating with multiple ZKP systems

isekai generates R1CS from source code in J-R1CS json format [Dre19]. Integration with ZKP systems is done via a library that reads the file and converts it for the interface used by each ZKP library. We have built two of such libraries, one in C++ and one in Rust, so that interfacing with the ZKP libraries is done easily using their native programming language.

The ZKP systems that we integrate with are the following:

- libsnark, which gives [Gro16] and [BSCTV13] schemes
- dalek using [BBB+17] scheme
- libiop for [AHIV17] and [BSCR+18] schemes

5.2 Benchmark setup

We have compared all the ZK schemes supported by isekai using identical arithmetic circuits. This gives consistent results, although there may still be some implementation issues, compiling options or system settings (such as curve choice) that can affect the results.

The computer used was a Lenovo T580 laptop with an i7–8550U processor (base frequency @ 1.80 GHz, turbo @ 4.00 GHz), 32 GB RAM and a 1 TB SSD hard drive. Note that for computations involving the most constraints, the entire memory was used and the system had to swap, which obviously affected performance.

The results are plotted against the number of constraints.

5.3 Primitives

We used 3 primitives, with different input sire, as input for the benchmark. We call them conditional, median and hash. They are summarized in the table below.

The first computation is simply doing extensive *dynamic memory access* (think of a[b[i]]). Internally, isekai uses many conditionals (ifs) to handle this, and the number of conditions grows linearly with the array size. The tests are named **cond10**, **cond100** and **cond1000**, with array sizes 10, 100 and 1000 respectively. Cond1000 has more than 6 million constraints. This shows the limitation of the current implementation, even if the benchmark was run before we implemented the optimisations of section 4.3 and does not benefit from it. The current version of isekai generates 66043 and 4267258 constraints, for cond100 and cond1000 respectively, a 22% and 32% improvement, getting close to the theoretical 33% (from 3n to 2n).

The second computation is simply *sorting an array and returning the median.* The sorting involves many comparisons that are not ZKP friendly. As a result, we could not perform the test of sorting an array of 1,000 elements because it generated more than 5 million constraints. This once again shows the limitation of the current implementation. The tests are named **med10** and **med100** with array sizes of 10 and 100 respectively.

Finally, for the third test, we have selected a widely-used function: a *sha256 computation.* While the first two tests show isekai's limitations, this one shows how powerful isekai can be. To implement a zero-knowledge proof of a sha256 computation, we simply took the first C++ implementation we

Table 1: Circuit and R1CS

Name	**Circuit***	**R1CS***	**Constraints**	**Witnesses**
		Dynamic memory access		
cond10	0.2	0.4	1,767	1,745
cond100	6.7	40.8	85,745	85,461
cond1000	600.0	4,181.0	6,266,234	6,263,354
		Sorting an array		
med10	0.2	0.4	4,752	4,646
med100	24.1	77.2	506,502	495,326
		sha256 computation		
h32	4.2	55.6	41,210	40,958
h128	13.0	502.0	145,061	144,403
h512	43.3	2,640.0	481,091	478,906
h1024	85.5	7,060.0	904,058	899,843

** size in MB*

found on the web and modified it slightly to make it compatible with isekai — the changes were easy and straightforward. Then, using isekai, we were immediately able to produce a proof of a hash computation for all proof schemes. The tests are named **h32**, **h128**, **h512** and **h1024** and compute the sha256 of a byte array of size 32, 128, 512 and 1024, respectively. h1024 generates around 1 million constraints.

5.4 Proof Time

We start by looking at proof times. We see that Ligero has a very good performance, comparable to those of zk-SNARKs even without a trusted setup. However, it did not work on the last two tests, those with many constraints, for which we kept getting an 'out of memory' error. This may be due to the implementation, because we had to implement some padding regarding the number of variables and this has an impact on both performance and stability.

The graph does not always show an increase, meaning that the number of constraints is not the only factor affecting performance. The complexity of the constraints is also important.

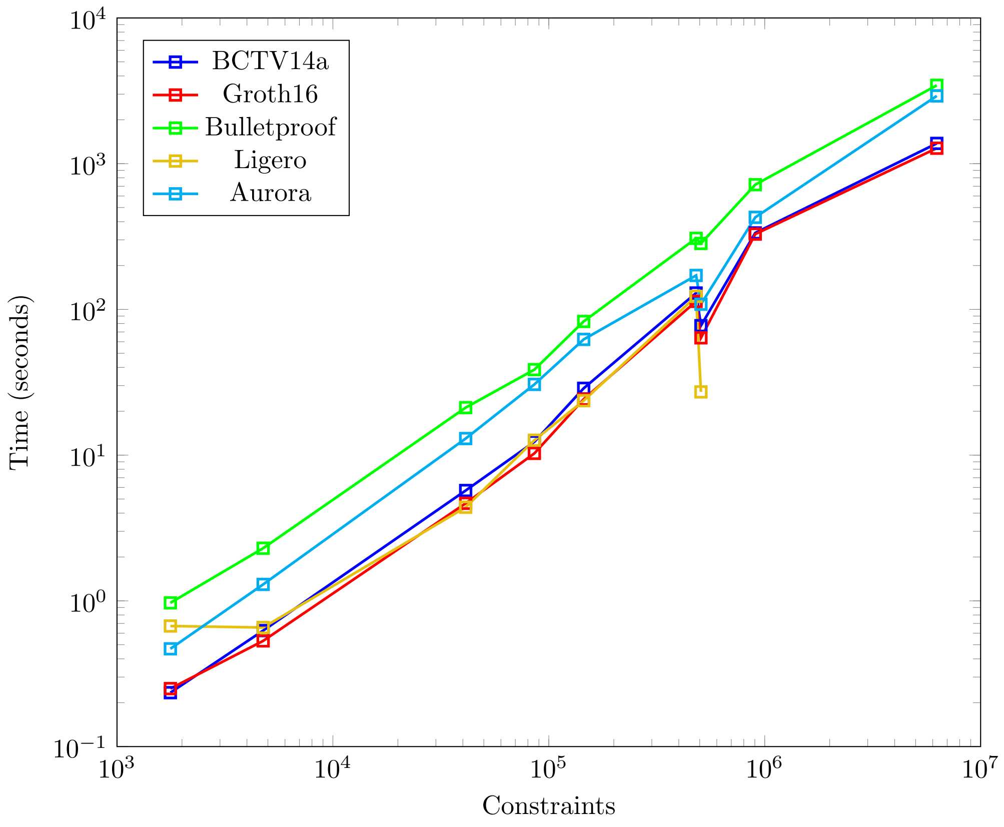
5.5 Results: Proof and Trusted Setup Sizes

Next, we look at proof and trusted setup sizes. Bulletproof has the lowest performance overall in the previous chart but we see here that its proof size stays really low, considering that it does not require a trusted setup. This leaves room for a trade-off between proof size and proof time.

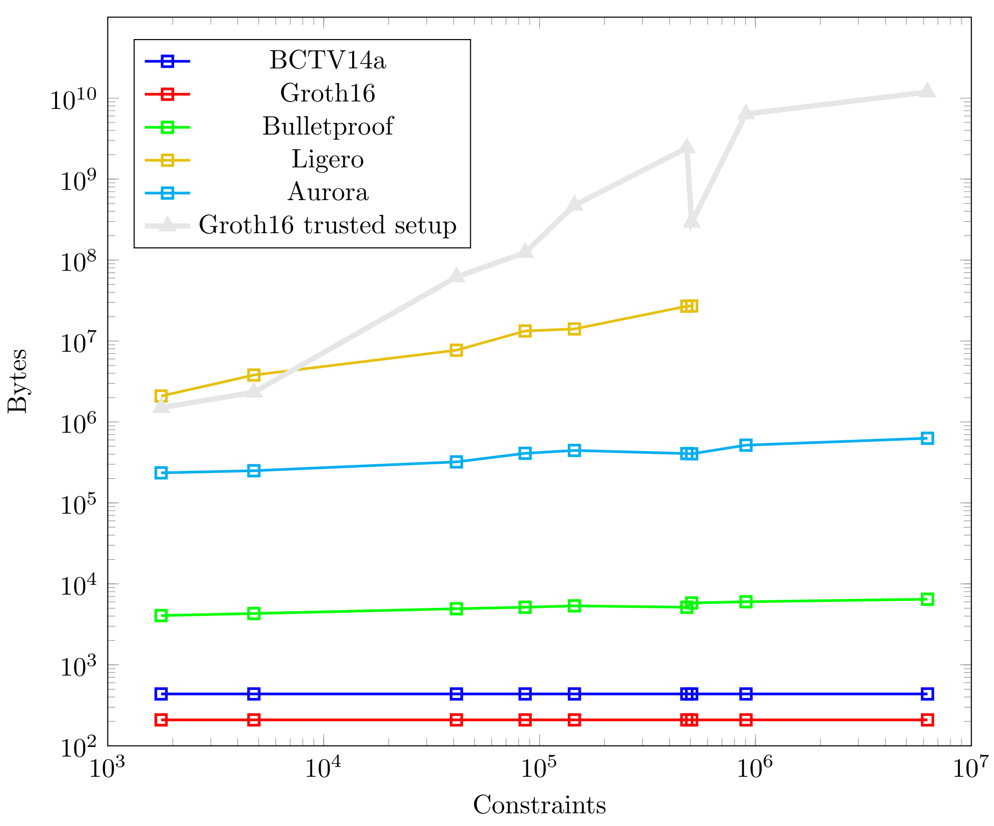
The proof size of zk-SNARKs is extremely small and constant: 209 bytes for the Groth16 scheme and 437 bytes for BCTV14a. However, the size of the trusted setup grows rapidly, up to 12 gb for 6 million constraints. Note that this is the raw data, and it turns out that this data compresses very well. One can expect a 10:1 ratio when compressing a trusted setup.

We also see that Ligero's performance comes at a cost: its proof size is much higher than that of Bulletproof (4 to 6 kb) and Aurora (200 to 600kb).

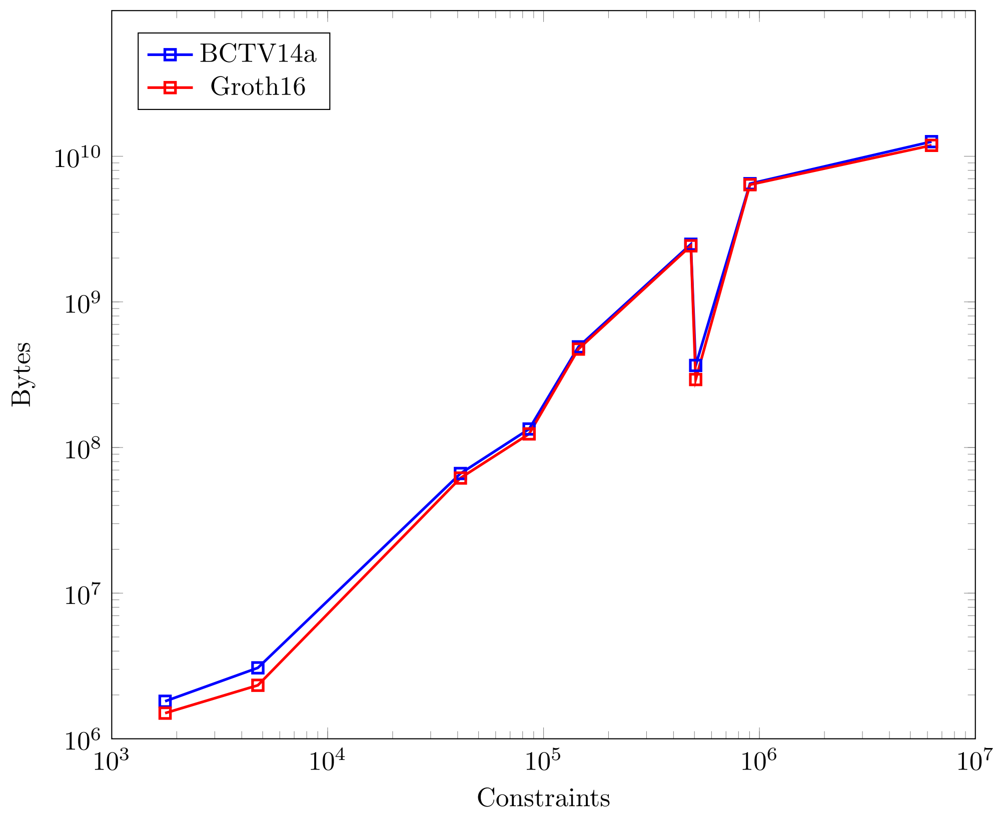
Plot 1: Proof Time



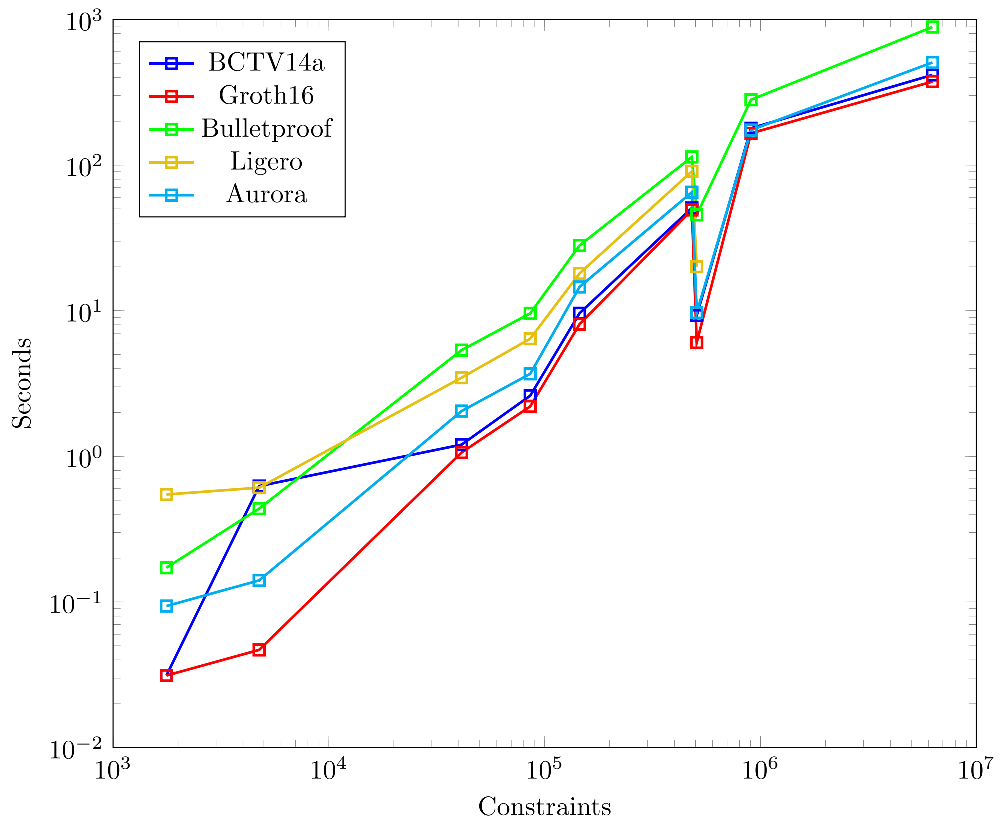
Plot 2: Proof Size vs. Trusted Setup



Plot 3: Size of Trusted Setup



Plot 4: Verification Time



5.6 Results: Verification Time

Finally, we look at the verification time, which is shown in Plot 4. It follows the same pattern as the proving time but is always significantly faster. As noted before, dynamic memory access is currently not handled efficiently in isekai, which results in slow verification times for the **cond1000** computation.

5.7 Conclusion

In conclusion, there is no overall winner: all proof systems have different pros and cons, different properties and different security assumptions. As long as issues related to trusted setup size and generation are handled, zk-SNARKs clearly have the best performance. Bulletproofs manage an impressively small proof size without trusted setup, and Aurora has a faster proof time and is quantum resistant.

6 Summary and future work

We explained the need for compiling computer programs into the low-level language used by zero-knowledge systems and how we implemented an LLVM fronted. As the main contribution of this paper, we identified problems for interfacing with LLVM and implemented solutions to them. Moreover we showed how isekai can be used for consistent benchmarking of different zero-knowledge proof schemes.

During this project we identified several directions of future work. We would like to add support for additional high level languages having LLVM support, for example Rust. We would like to integrate with more ZKP systems and implement the optimisations we found. Finally we would also like to improve LLVM integration to enable recursion and pointer casting, identify LLVM optimisations that do not break our decompilation algorithm, look at memory handling of [BSCG+13], implement a WebAssembly frontend and support FHE and MPC schemes.

References

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2087–2104. ACM, 2017.

[BBB+17] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Report 2017/1066, 2017. `https://eprint.iacr.org/2017/1066`.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems*

Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013, pages 341–357. ACM, 2013.

[BJ66] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. `https://eprint.iacr.org/2018/046`.

[BSCG$^+$13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. Cryptology ePrint Archive, Report 2013/507, 2013. `https://eprint.iacr.org/2013/507`.

[BSCR$^+$18] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. Cryptology ePrint Archive, Report 2018/828, 2018. `https://eprint.iacr.org/2018/828`.

[BSCTV13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. Cryptology ePrint Archive, Report 2013/879, 2013. `https://eprint.iacr.org/2013/879`.

[CFH$^+$14] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. Cryptology ePrint Archive, Report 2014/976, 2014. `https://eprint.iacr.org/2014/976`.

[CHM$^+$19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Report 2019/1047, 2019. `https://eprint.iacr.org/2019/1047`.

[COS19] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. Cryptology ePrint Archive, Report 2019/1076, 2019. `https://eprint.iacr.org/2019/1076`.

[DK19] Guillaume Drevon and Aleksander Kampa. Benchmarking zero-knowledge proofs with isekai. `https://sikoba.com/docs/SKOR_isekai_benchmarking_201912.pdf`, December 2019.

[Dre19] Guillaume Drevon. J-r1cs : a json lines format for r1cs. `https://www.sikoba.com/docs/SKOR_GD_R1CS_Format.pdf`, April 2019.

[ET18] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/-GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*, pages 1084–1091. IEEE, 2018.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptology ePrint Archive*, 2016:260, 2016.

[GT83] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 246–251. ACM, 1983.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. `https://eprint.iacr.org/2019/953`.

[HBHW19] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specifications, 2019. `https://eprint.iacr.org/2017/1066`.

[KZM$^+$15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. `https://eprint.iacr.org/2015/1093`.

[PGHR13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Report 2013/279, 2013. `https://eprint.iacr.org/2013/279`.

[WSH$^+$14] Riad S. Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. Cryptology ePrint Archive, Report 2014/674, 2014. `https://eprint.iacr.org/2014/674`.

[Zak11] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In Cristina Videira Lopes and Kathleen Fisher, editors, *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 301–312. ACM, 2011.