Using zkMIPS, a client can outsource a program written using the MIPS instruction set, have it executed, and in addition to getting the result, receive an easily verifiable proof that this result is correct. The software component that creates this proof is called a zkVM

.

The input to the zkVM is the execution trace

of the program execution. Since a zkMIPS computation happens in steps, its overall state can be defined by the values of a finite list of variables. A valid computation is a sequence of states, from a well-defined initial state to some final state, in which each state transition represents a valid computation step. This sequence can be represented as a table whose columns represent the list of variables and whose rows represent each step of the computation. The following diagram shows a toy example.

[

Screenshot at 2023-08-21 11-34-40

1207×462 17.4 KB

](https://ethresear.ch/uploads/default/original/2X/f/febfc43426d9f7446b58fc79e35899d11cf8affd.png)

Read Jeroen van de Graaf's previous article utilizing this toy example.

To prove that the result of a program is correct, one has to prove that all transitions from each state (line) to the next is valid, i.e. that it's consistent with the opcode of that particular instruction of that program step. So for each different instruction this consistency proof is different, resulting in an arduous task for zkVM software developers.

However, there is good news. Last week, Justin Thaler and Srinath Setty, together with Arasu Arun and Riad Wahby published a new, more efficient approach to perform these consistency checks on a16zcrypto's website. They published two papers presenting two complementary systems: Jolt and Lasso. To understand their approach it is easiest to start with Jolt.

To start, we know that a Boolean function can be represented by a truth table.

x

y

z = x AND y

0

0

0

0

1

0

1

0

0

1

1

1

Likewise, instead of bits, we could use bytes, to get a table like

x

y

z = x AND y

00000000

0000000

00000000

00000000

0000001

00000001

\vdots

\vdots

\vdots

11111111

1111110

11111110

11111111

1111111

11111111

Note that we have 2^8 possible inputs for x, and also for y, so the overall table size is 2^{16} = 65536 rows.

If instead of bytes, we use 32 bit words, the overall table size will be $(2^{32})^2 = 2^{64}$ rows; and in the case of 64 bit words, the table would have 2^{128} rows.

This number, 2^{128}, is gigantic. In fact, the AES encryption algorithm uses a 128-bit key, since it is considered infeasible to exhaustively search all possibilities. Likewise, implementing a table with 2^{128} rows in reality is completely impossible. This problem is addressed in the second paper, in which Lasso is presented. But don't worry about that for now; just assume such gigatables exist.

Under this assumption we can implement a (bitwise) AND between x and y, each of 64 bits, by using a gigatable T; and we would get: $z = T[x,y]$.

Note that zk proofs are slightly different. We are not computing z = AND

(x,y) since the Prover gets x,y and z already as inputs (where probably z was computed before, using an AND instruction). Instead, the Prover needs to show that these values are correct. In other words, he has to PROVE that $z \equiv AND\ (x,y)$, where we used a special equality symbol for emphasis.

The basic idea behind Jolt+Lasso is to verify an execution trace by using table lookups. These are also implemented using polynomials, but simpler ones than those used by Plonk or HALO2, leading to a 10x speedup or more for the Prover. Unfortunately this comes at a cost of a 2x delay for the Verifier.

Now the opcode we used in this example is AND. But it is clear that we can represent any other opcode in a table: bitwise OR, bitwise XOR, ADD, SUB etc. Each opcode will have a different corresponding table, of course.

The Jolt paper does not just discuss opcodes. It presents an abstract but equivalent model of a CPU, and proves that each instruction of its Instruction Set Architecture (ISA) can be implemented using table lookups, assuming the existence of gigatables.

The details of Jolt are sometimes a bit tedious, but nothing highly abstract or mathematical. It has the flavor of CPU design, but using the specific building blocks that the Lasso model offers, not the tools of an electrical engineer.

To give an idea of the topics that Jolt discusses, here are a few examples:

- To verify an AND opcode on two 64-bit words, one doesn't need a gigatable. Instead, one can split each word in 8 bytes, and apply 8 bitwise ANDs, on each pair of bytes. This works, because in an AND opcode the bit positions are independent.
- To verify an ADD opcode, one can take advantage of the addition in the finite field, provided that its size is larger than 2^{64}. Also, one must deal with overflow exactly the way that the CPU does.
- Division is an interesting case. If z = DIV

(x,y), Jolt will not try to replicate this computation. Instead it will verify that $x = z \cdot y$, and provide a proof thereof. More

precisely, if also w = MOD

(x,y), then Jolt will verify the quotient-with-remainder equation x = z $*$ y + w.

- In order to speed up verification, we are allowed to add as many extra variables (registers) as we like, as long as they don't interfere with the CPU's logic. By the way, floating point operations are not provided.

The Jolt paper discusses all instructions included in the CPU's ISA, providing a detailed description of how each instruction can be reduced to table lookups, some of them simple, others using gigatables.

The real hard part of this approach is the implementation of these gigatables. Most values in these tables will never be accessed, they exist only conceptually.

How this is possible is described in the companion Lasso paper. This will be the topic of our next post, coming soon!