Designing efficient R1CS circuits

Daira Hopwood
daira@jacaranda.org
@feministPLT
https://github.com/daira/r1cs
@daira on chat.zcashcommunity.com

Zero knowledge proving systems

- A *statement* is a proposition we want to prove. It depends on:
  - *Instance variables*, which are public.
  - *Witness variables*, which are private.
- Given the instance variables, we can find a short *proof* that we *know* witness variables that make the statement true, without revealing any other information.
- A proof of *knowledge* is stronger and more useful than just proving the statement is *true*. For instance, it allows me to prove that *I know* a secret key, rather than just that *it exists*.
- The proof can be just a string; anyone can verify it without interacting with the prover.
- I'm glossing over some details, such as *setup* and variations of the security properties, which are not the focus of this talk.

ZK proving systems in the real world

- Since ~2013, zk proving systems have become practical for real-world applications.
- Example: Zcash (https://z.cash)
  - Private Bitcoin-like cryptocurrency, with hidden amounts, senders and recipients.
  - (Simplified.) "I know the private key that shows I own *n*, which is a valid note with nullifier *nf* and a value that balances this transaction."

    Ensuring that a nullifier is not repeated prevents double spending.
- But… *only just* practical:
  - e.g. proof for a private payment at the initial launch of Zcash took > 40 seconds (reduced to 2.5 seconds using some of the techniques described later in this talk).
- This talk isn't about Zcash, but it shows the kind of things we want to be able to prove.

ZK proving systems in the real world

- The current focus is on proving cryptographic protocols are followed correctly.
- What kinds of things are used in cryptography?
  - Hash functions: $R = H(B)$
  - One-way functions: $Q = [x]\ P$
  - Building blocks: B is a bit string; $0 \le x < y$; P is a valid elliptic curve point; arithmetic; boolean logic; conditionals; …
  - Recursive validation: π is a valid proof for instance X.

Languages for statements

- In the future, statements will be written in high-level languages (e.g. Snarky, Zokrates).
- This talk is not about how to express statements in a concrete programming language. For that see:
  - https://z.cash/blog/bellman-zksnarks-in-rust/ for bellman (Rust library; used by Zcash)
  - https://o1labs.org/blog/posts/snarky.html for Snarky (O'Caml embedded DSL; used by Coda)
  - https://github.com/Zokrates/ZoKrates for ZoKrates (dedicated language; used in the Ethereum community).
- All of these systems compile to a language called R1CS (Rank 1 Constraint Systems), which is the subject of this talk.

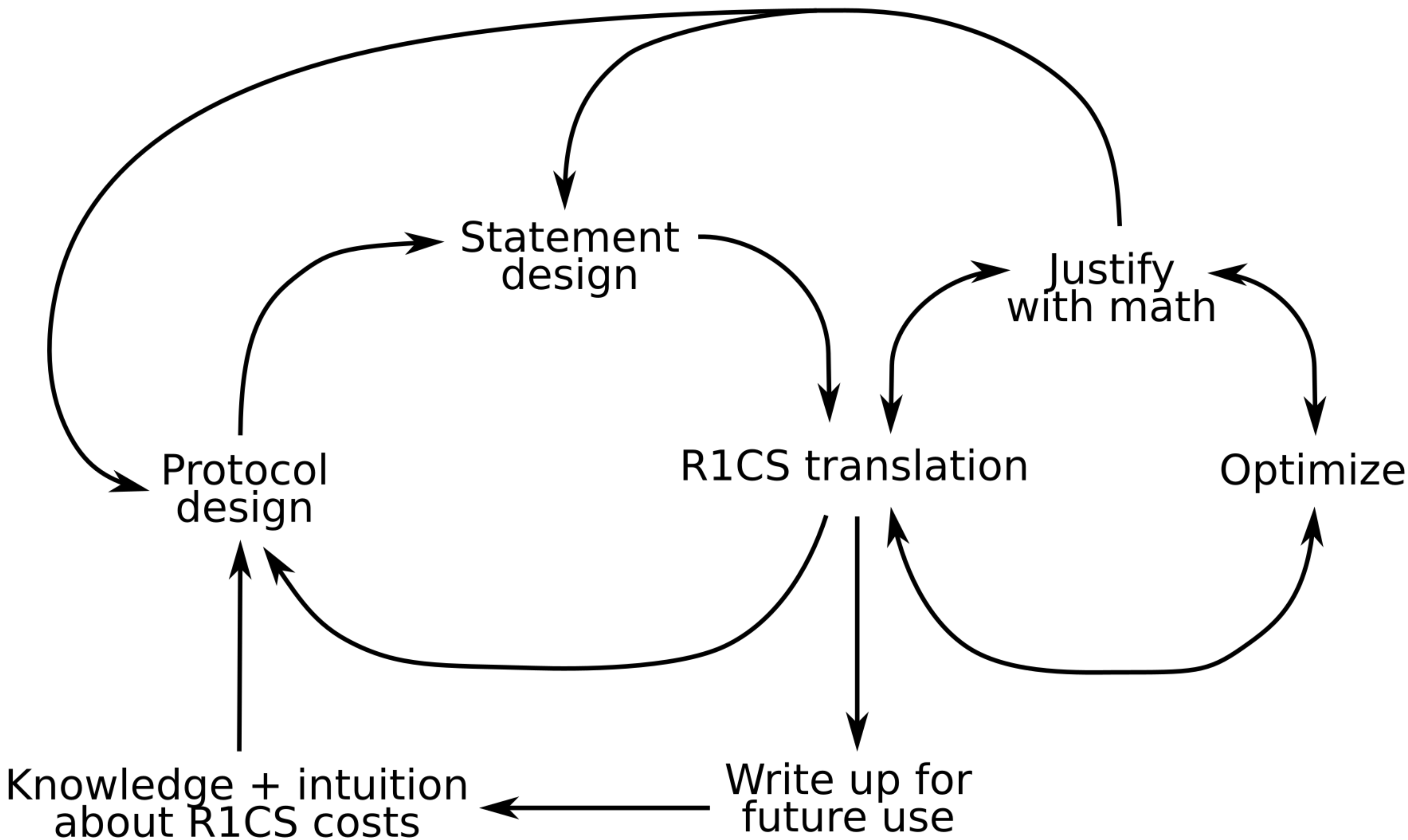
R1CS

- Even when programming in a higher-level language, it's necessary to know R1CS.
  - just as understanding the machine model is useful for writing efficient code in conventional programming languages…
  - but even more so, because Snarky, Zokrates, etc. expose many details of the R1CS model (and future proof-oriented languages are also likely to do so).
- We call R1CS programs *circuits*. This talk aims to give a flavour of how R1CS circuits are written and optimized.
- Many different proving systems use R1CS, and it's a current focus of standards development, so this knowledge is transferrable between systems.

R1CS is only a compilation target

- It is untyped (all values are field elements).
  - Always use types in specifications or when using a proving library / DSL.
- It has no direct support for abstraction.
  - It would be difficult to reverse-engineer the intended statement from an R1CS description. Use abstraction mechanisms supported by the proving library / DSL.
- It’s very error-prone.
  - Overflow, missing constraints, unhandled exceptional cases, etc. can cause silent fatal security flaws.

A possible design flow



Fields

- We have instance and witness variables. Variables have values in a *field*.
- A field supports addition, subtraction, multiplication and division of elements, with the following laws:
  - *associativity*: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ and $(a + b) + c = a + (b + c)$
  - *commutativity*: $a \cdot b = b \cdot a$ and $a + b = b + a$
  - *identities:* $a + 0 = a$ and $a \cdot 1 = a$
  - *inverses:* $a + (-a) = 0$ *and* $a \neq 0 \Rightarrow a \cdot (1/a) = 1$ (we write $a \cdot (1/b)$ as a/b)
  - *distributivity:* $a \cdot (b + c) = a \cdot b + a \cdot c$
- Examples: real numbers, complex numbers, integers modulo a prime.
- We use *finite fields* for cryptography, because elements have “short”, exact representations. In this talk, we only need integers modulo a prime: $\mathbb{F}_p$.

Consequences of using $\mathbb{F}_p$

- Field elements are *not* integers.
- We can use them to *represent* integers, if we're careful.
- "Short" means ~255 bits
  - which is enough to represent a lot of integers, but
  - we **always** need to be careful of overflow.
- We can also use them to represent bits: 0 or 1.
  - this is inefficient (but often necessary)
  - because the proving system still operates on the full field width.
- We can use them to represent themselves!
  - very useful for elliptic curve cryptography
  - but only if the field matches the prime we need.

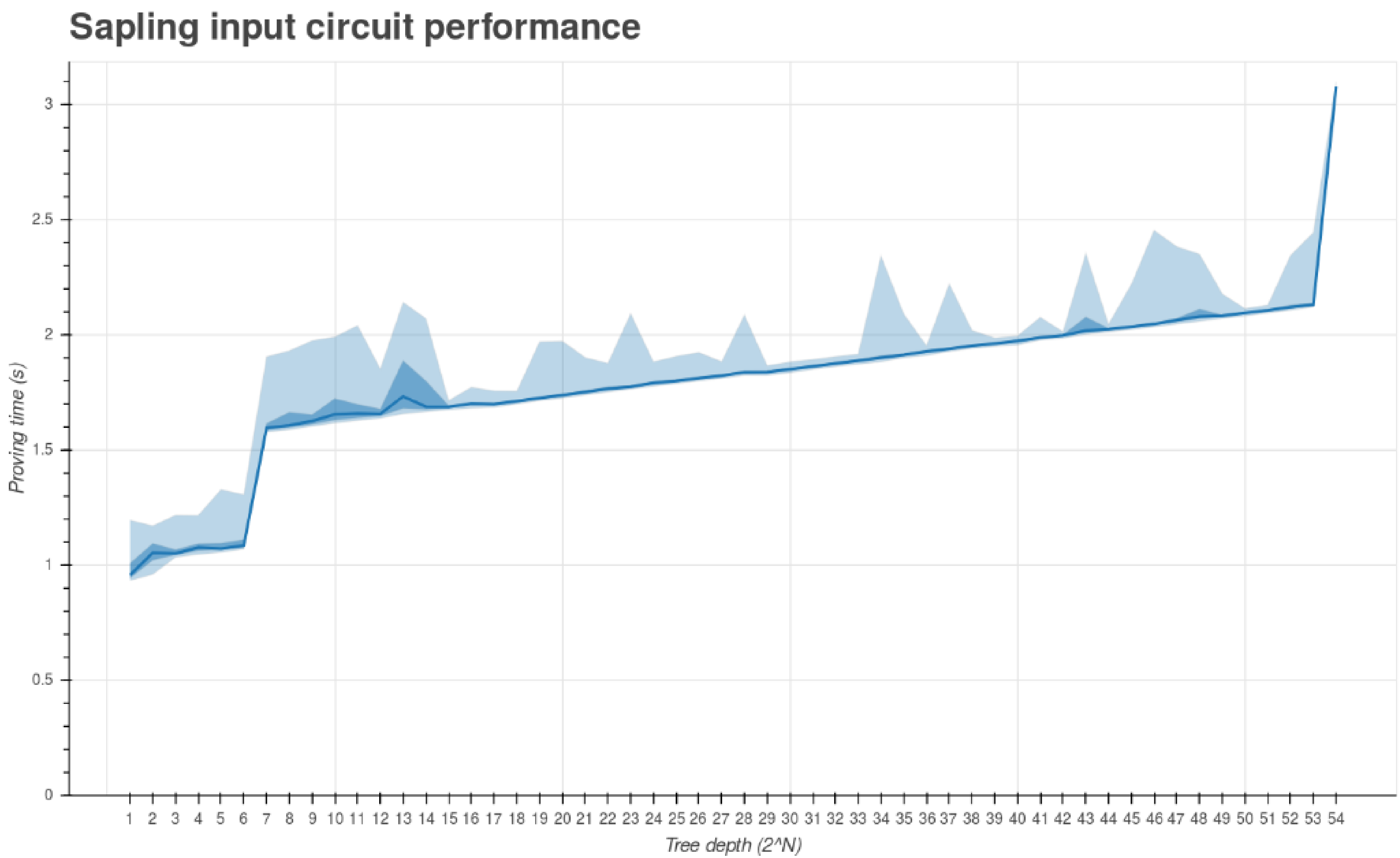
Rank 1 constraints

- A *rank 1 constraint system* is a set of *rank 1 constraints*, each of the form:

  $(A) \times (B) = (C)$

  where (A), (B), (C) are each *linear combinations* $c_1 \cdot v_1 + c_2 \cdot v_2 + \ldots$
- The c_i are *constant* field elements, and the v_i are instance or witness variables (or *1*).
- By convention we write "$\times$" for the multiplications that we need to count, but it's the same operation as "$\cdot$".
- Think of general multiplications and divisions as costing 1 constraint; additions, subtractions and scaling by constants are "free".
  - This is not quite accurate but still a good mental model for optimization. The cost of the circuit will be roughly dependent on the number of constraints (*not* the complexity of the linear combinations).

Dependence of proving time on circuit size



Rank 1 constraints

- R1CS is a *constraint language*.
- The inputs and outputs of a subcircuit are not predetermined.
  - $(A) \times (B) = (C)$ doesn't mean (C) is computed from (A) and (B), just that (A), (B) and (C) are consistent.
  - more generally, an implementation of "$x = f(a, b)$" doesn't mean that x is computed from a and b, just that x, a, and b are consistent.
- Constraint languages can be viewed as a generalization of functional languages:
  - everything is referentially transparent and side-effect free
  - there is no ordering of constraints
  - composing two R1CS programs just means that their constraints are simultaneously satisfied.

Correctness and efficiency

- Multiplication and linear combinations allow us to represent arbitrary circuits:
  - $(1 - b) \times (b) = 0$ is a *boolean constraint* for b.
  - "a AND b" can be implemented as $(a) \times (b)$, and "NOT b" as $1 - b$.
  - This is a complete set of boolean/bit operations.
- The question is how to represent circuits
  - efficiently (roughly: in the fewest constraints), and
  - correctly (expressing what we intended).
- Correctness is a prerequisite for security. It is not sufficient (we also need to be implementing a secure protocol), but it is necessary.
- Efficient use of fields can allow *4 orders of magnitude* improvement over naive use of bit operations.
  - multiplying two 255-bit numbers would require ~34000 bit operations, but we can do it in one constraint.

Starting with the basics

- We've already seen $(1 - b) \times (b) = 0$.
- This is an instance of a common pattern:

  $(A) \times (B) = 0$ implements "$A = 0$ or $B = 0$".
- We can substitute $A = P - Q$ and $B = R - S$, to get "$P = Q$ or $R = S$".
- We did not "reify" $A = 0$ and $B = 0$ as boolean variables and explicitly implement OR.
  - Don't reify constraints as booleans unless you have to. There is a way to do that, but it's complicated (and costs 3 constraints).
- This isn't the only way to do a boolean constraint: $(b) \times (b) = (b)$ also works.
  - It's useful to be able to recognise alternative ways of doing the same thing when reverse engineering R1CS circuits written by others / generated by other libraries.

Inequalities

- What about $A \neq 0$?
- 0 is the only field element that doesn't have a multiplicative inverse ($1/0$ does not exist).
- So $(A) \times (A_{inv}) = 1$ ensures that $A \neq 0$.
- We've added a witness variable, A_{inv}, that is just an implementation detail rather than part of the original statement. This is very common.
  - It is a witness variable, not an instance variable, because it would leak information, in this case the value of A, if made public.
- $A - B \neq 0$ is equivalent to $A \neq B$. From now on we'll take equivalences like this as obvious.

Division

- $a = c/b$ is equivalent to $(a) \times (b) = (c)$.
- What does $(a) \times (b) = (c)$ do when $b = 0$?
  - It constrains c to 0 and leaves a unconstrained.
  - This makes sense, but is probably not what you want. So don't do that (either constrain or prove $b \neq 0$).
- Notice that division is the same cost as multiplication. This is different from computing inverses in a prime field "outside the circuit", which is much more expensive than multiplication.
  - Technically, you still need to compute the division when proving. But the cost of that is far outweighed by the per-constraint cost of proving.
  - The different relative costs of operations may lead us to choose different algorithms and representations.

Inversions

- Division being expressed via multiplication is a special case of a general principle: inversions are easy to express.
- If f is invertible, $y = f^{-1}(x)$ is equivalent to $f(y) = x$.
- (In the previous slide, $f^{-1}(x)$ was x/b.)

Boolean operations: AND

- Let's use n-ary AND as an example.
- How many constraints do we need to implement $b = \text{AND}_{i \in \{1..n\}}\, a_i$?
- There's an obvious implementation in $n - 1$ constraints. Can we do better?
- We know the answer is boolean:

  $(1 - b) \times (b) = 0$
- If the answer is *1,* then all of the a_i must be *1*:

  $(n - \sum_{i \in \{1..n\}} a_i) \times (b) = 0$
- If the answer is *0*, then not all of the a_i must be *1*:

  $(n - \sum_{i \in \{1..n\}} a_i) \times (inv) = (1 - b)$
- So, at most 3 constraints independent of n.
- Notice how we're making use of the representation of booleans as *0* or *1* and doing arithmetic on them, in order to take advantage of "free" linear combinations. (This won't overflow because $n < p$.)
- n-ary OR can be implemented similarly.

Range constraints

- For binary ranges $a \in \{0..2^n\text{-}1\}$:

  boolean-constrain a_i for $i \in \{0..n\text{-}1\}$

  $(\sum_{i \in \{0..n-1\}} a_i \cdot 2^i) \times (1) = (a)$

- For $a \in \{0..c\text{-}1\}$, let n be the bit length of c and constrain $a \in \{0..2^n\text{-}1\}$ and $a + 2^n - c \in \{0..2^n\text{-}1\}$.
- There's a more efficient approach that depends on the bit pattern of $c - 1$. [Zcash specification, appendix A.3.2.2]
- The "$\times$ (1)" is technically redundant: we could perform a substitution to eliminate it. In general it's always possible to substitute linear combinations rather than adding another constraint.
  - Your constraint proving library/language should do the substitutions for you. bellman supports this; not sure about Snarky or ZoKrates.

Boolean operations: XOR

- $c = a$ XOR b can be implemented as:

  $(2 \cdot a) \times (b) = (a + b - c)$

  which is equivalent to $c = a + b - (a \text{ AND } b)\ ?\ 2 : 0$.

- What about n-ary XOR? How many constraints do we need to implement $b = \text{XOR}_{i \in \{1..n\}}\, a_i$?

- b is the least significant bit of $\sum_{i \in \{1..n\}} a_i$.

  boolean-constrain b_j for $j \in \{0..ceiling(lg(n))\text{-}1\}$

  $(\sum_{i \in \{1..n\}} a_i) \times (1) = (\sum_{j \in \{0..ceiling(lg(n))-1\}} b_j \cdot 2^j)$

- Now the answer is b_0.

- So, at most $ceiling(lg(n)) + 1$ constraints independent of n.

Conditionals

- A selection constraint $(b\ ?\ x : y) = z$, where b has been boolean-constrained, can be implemented as:

  $$(b) \times (y - x) = (y - z)$$

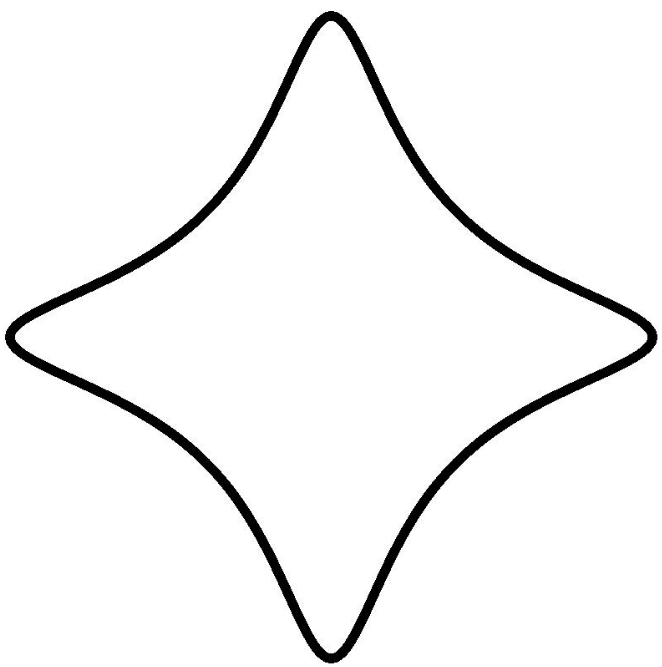
- We can see this is correct by case analysis on b:
  - If $b = 1$ then $y - x = y - z$ therefore $z = x$.
  - If $b = 0$ then $y - z = 0$ therefore $z = y$.

Elliptic curves

- The most commonly deployed public key cryptosystems use elliptic curves.
- With elliptic curves we can also implement collision-resistant hash functions.
  - Hash functions are the "nails" of cryptography, used everywhere.
- EC crypto can be very efficient in a circuit, compared to symmetric crypto:
  - SHA-256 takes ~27000 constraints.
  - A comparable elliptic curve Pedersen hash takes ~864 constraints, not including boolean-constraining the input.
  - This is because SHA-256 is mainly bit operations, while Pedersen makes full use of the field.
  - This is completely the opposite situation to crypto "outside the circuit".

Edwards curves

- Equation of a circle: $u^2 + v^2 = 1$
- Equation of an Edwards curve: $a \cdot u^2 + v^2 = 1 + d \cdot u^2 \cdot v^2$
- Over real numbers, the curve looks something like:



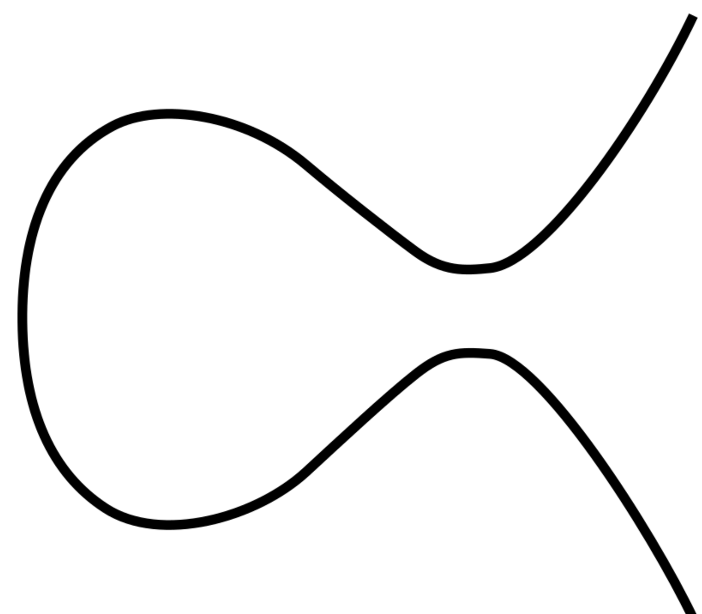
Edwards arithmetic

- The circuit implementation basically follows the textbook “affine” equations:
  - naive: 7 constraints for addition and doubling
  - optimized: 6 constraints for addition, 5 for doubling
  - Details in the Zcash protocol spec, appendix A.3.3.
- Circuit implementations of elliptic curve arithmetic are actually simpler than out-of-circuit ones, because field division is as efficient as multiplication.

Montgomery curves

- Each Edwards curve is "birationally equivalent" to a Montgomery curve:

$\leftrightarrow$



Montgomery arithmetic

- For a Montgomery curve, addition takes 3 constraints, and doubling takes 4 constraints
- … but the Montgomery addition doesn't work in all cases; we have to prove that the exceptional cases don't occur.
- We can use the birational equivalence to convert between the fast-but-tricky Montgomery curve, and the slower-but-easier Edwards curve.
- Best to leave this optimization to libraries that are thoroughly reviewed.
- Edwards scalar multiplication $[x]\ P$:
  - fixed P: 750 constraints, variable P: 3252 constraints.
- Optimized Montgomery scalar multiplication $[x]\ P$:
  - fixed P: 506 constraints, variable P: 2249 constraints (a third better).

Side rant about correctness proofs

- Common wisdom about use of proofs of (conventional) program correctness: “too hard”, “not ready for prime time”, “the tooling is not there”, “doesn’t scale to real-world programs”, “too hard to maintain when program changes”.
- No! DO PROOFS OR YOU WILL FAIL
  - Optimizations used in Zcash Sapling had proofs of correctness, so this *is* practical.
- Do not whine about needing to do proofs. If you can’t do them, ask a mathematician / cryptographer / appropriate expert. There is a cultural problem with viewing proofs as rocket science, don’t make it worse.
- You don’t necessarily need to use formal theorem provers.
- Do proofs about things that are non-obvious
  - to you, or to a reviewer
  - a lot of things are obvious because the constraint system directly matches the high-level specification.
- Typical proofs are of “this unhandled case can’t occur”, “these algorithms are equivalent”. They will mostly stay valid, or be adapted easily, for changes in the lower-level detail of the constraint system.

Recursive validation

- Suppose we want to validate a zk proof inside a circuit.
- This allows proving a tree of computation of arbitrary size, in just one proof with constant size and validation time.
- This is a bit of a tour de force and requires much more math than we have time for. The key component is a “pairing”, which is another kind of elliptic curve algorithm.
- Pairings can be implemented in ~7000 constraints, and a full validation (for Groth16) in ~25000 constraints (i.e. less than SHA-256).
  - These are preliminary numbers for a specific curve that might not be quite secure enough, but further optimizations are possible.

Conclusions

- Writing R1CS programs is interesting and fun…
- but very error-prone. Better tools will be needed.
- Huge performance gains are possible by choosing the right algorithms, representations, and fields.
- Do proofs of correctness where needed.

- These slides are at https://github.com/daira/zcon0/blob/master/zkproofs.pdf

Bonus slides: optimization

- Find equivalent expressions of algorithms and use the one with the fewest constraints.
- If expressions are equivalent except for corner cases:
  - constrain the corner cases not to occur, or
  - (better, because no extra constraints) prove that they never occur.
- Switch between multiple representations.
- Change the higher-level protocol to avoid/minimize use of expensive primitives.
- Find non-optimizable things first. Try to reuse values that are unavoidably needed.
- Use algebraic rearrangement to find common subexpressions / make the remaining computations linear.
- Linear expressions are (almost) free. If you are left with linear constraints, remove them by substituting into uses.
  - Ideally, your proving library API / DSL should make this easy.

Bonus slides: optimization

- Merge to do two things at once.
  - Example: merging with boolean constraints in constant comparisons (Zcash spec A.3.2.2).
- Specialize for constants.
  - Example: lookup from a constant window table in fixed-base scalar multiplication.
- Use nondeterminism.
  - Examples: proving that a value is a square, or non-zero.
- We have concentrated on minimizing number of constraints, but there is also a cost to computing the witness. This can often be optimized by combining operations.
- n-ary operations can often be made less than n times as expensive as 2-ary.
- Trade operations inside the circuit for operations outside.
- Booleans are (typically) represented as field elements and you can do non-boolean arithmetic on them.
- The most efficient operations are those you can remove.

Bonus slides: opinionated advice

- Avoid 90s crypto
  - hashes before SHA-256; ciphers before AES.
  - They tend to be inefficient, particularly so in a circuit, even before considering security.
- Many standardized algorithms incur expense that is unnecessary for the small fixed input sizes typically used in circuits.
  - e.g. can use BLAKE2s on a single block directly as a PRF, no need for HMAC/HKDF
  - check with a cryptographer if you are not one.
- Scour the cryptographic literature for cheaper primitives (maybe discarded because they weren't competitive in out-of-circuit computation).
- Use personalization. It's typically free or very cheap, and prevents some chosen-protocol and replay attacks.
- Minimize the primitives used. Circuit programming is difficult and the fewer distinct primitives, the less chance of mistakes and the easier review will be.
- But don't be afraid to specialize if it really helps performance.
- Include redundant checks if they simplify the security analysis and are cheap enough.
- Don't spend time optimizing stuff that makes little difference to overall performance. "Premature optimization is the root of all evil" still applies.
- Set a well-defined "good enough" criterion and stick to it.
- If you don't have imposter syndrome about designing zk circuits in 2018, you're probably doing something wrong.