Daira Hopwood
@feministPLT <daira@z.cash>
@daira on chat.zcashcommunity.com
Zcon0, Montreal, 27 June 2018

Layering of a proof system

Statements

- What are you trying to prove?
  -- For given x, I know a witness w, such that P(x, w).
- Always use types
  -- For given x : X, I know a witness w : W, such that P(x, w).
- Types allow the proving library API or DSL to catch as many low-level mistakes as possible.
- Examples
  -- For given h : Byte[32], I know w : Byte[64], such that BLAKE2s("Zcon0_ex", w) = h.
  -- For given Merkle tree root rt : Hash, I know a leaf and path (leaf : Hash, path : Hash[Depth], pos : Nat) in the tree rooted at rt
  -- For given pk : Point, I know sk : Scalar such that [sk] G = pk.
- Statements are composable while hiding intermediates
  -- E.g. For given rt : Hash, I know (w : Preimage, leaf : Hash, path : Hash[Depth], pos : Nat) such that H(w) = leaf and (leaf, path, pos) is in the tree rooted at rt.
- The proving system is a black box (nearly). You can design statements (almost) independently of knowing how it works.

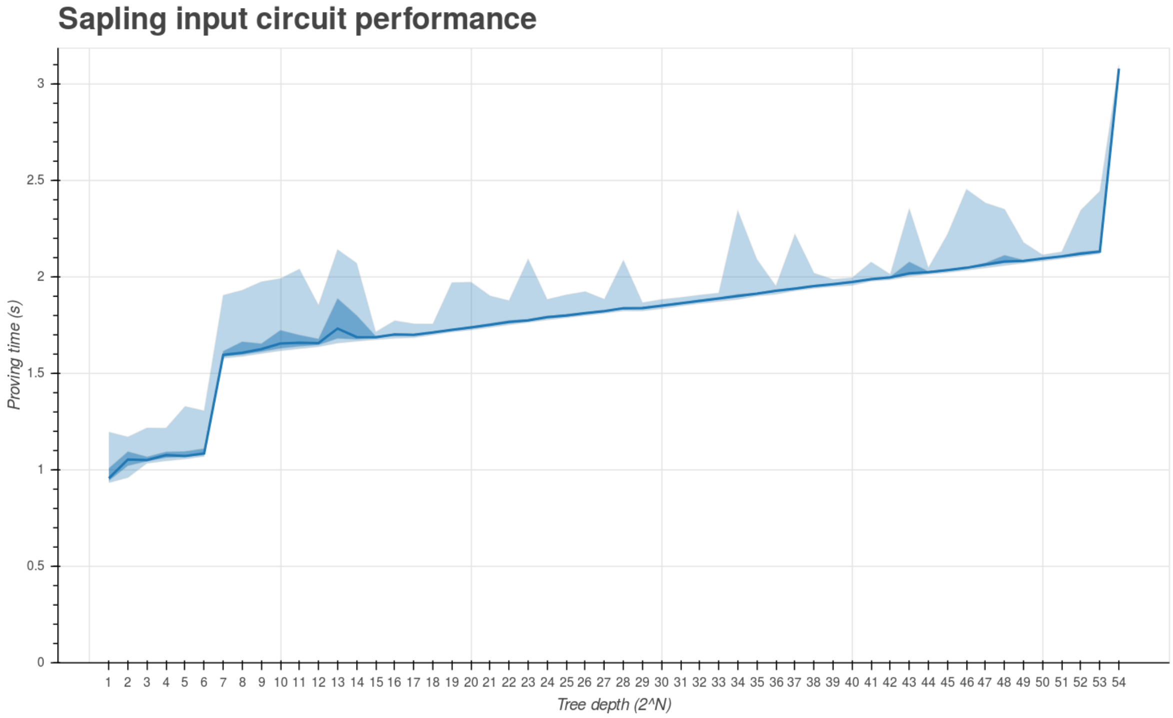
Ok, but how do we express statements?

- For this session: Rank 1 Constraint Systems.
- Applies to PHGR13, Groth16, Bulletproofs, bunch of others. Reusable knowledge.
- Set a finite field F. All finite fields are GF(p^m). For this talk, we focus on F = GF(p).
  -- GF(2^m) is underexplored.
- We have a set of variables <u>x</u> : <u>F</u>, <u>w</u> : <u>F</u>. The R1CS is defined by constraints <u>(A) x (B) = (C)</u> where A, B, C are linear combinations $a_0.u_0 + a_1.u_1 + \ldots$
- This is complete for bounded statements.
- How do we express a given statement *efficiently*?
- How do we design statements that we can express more efficiently?

Arithmetic circuit
    v ← don't need this (for now)
R1CS
    v ← somebody else's problem
QAP…

- By designing at the R1CS level, we expose the main determinant of proving efficiency: number of constraints.

- R1CS programming is low-level, but *not* like assembler -- more like an esoteric language.
- Graph of proving time vs circuit size [thanks to @str4d]: ~linear with sharp steps at powers of two.



- Verification time for SNARKs has some dependence on instance size, but can use hashing trick, so effectively O(1).

Circuits are *constraint* programs

- $y = x^2 \longleftrightarrow x = +/- \text{sqrt}(y)$ (if it exists)
- $y = H(x) \longleftrightarrow x$ is an H-preimage of y (and prover knows it)
- $y = E_K(x) \longleftrightarrow D_K(y)$ (and prover knows K)
- $q = a/b \longleftrightarrow q.b = a$ (if the inverse exists)
- what if a = b = 0? then q is *unconstrained* (often, but not necessarily, a design error).

Relative costs are very different from outside computation

- outside: I ~= 100 M, inside: I = M
- outside: AND < 0.0001 M, inside: AND = M
- outside: is_bool ~= 0, inside: is_bool = M :-(
- inside: m = 0, a = 0
- ⇒ reevaluate performance trade-offs
- Examples:
  -- favours asymmetric crypto relative to symmetric
  -- more generally, favours algebraic crypto in F relative to “bit twiddling”, because treating single bits as F elements is inefficient
  -- elliptic curve arithmetic: favours affine coordinates, not projective
  -- fixed-base mult gets even faster relative to variable-base (more generally: specializing for constants works well)

-- some things don't change: birationally equivalent twisted-Edwards/Montgomery curves still rock

- Concrete examples:
  BLAKE2s 21136 M, SHA-256 compression ~27534 M
  Pedersen hash (Bit[510] → F w/ Sapling optimizations) 1369 M
  MiMC (255-bit F[2] → F) 640 M
- Jubjub Montgomery scalar mult: fixed-base 506 M, variable-base 2249 M.
  https://github.com/zcash/zcash/issues/2230#issuecomment-361063268
- Not all of these scalar mult optimizations are used in Sapling due to complexity (fixed base is used in Pedersen hashes)

Deep dive: elliptic curve arithmetic

- Picture of Montgomery curve over ℝ (for intuition only)
- Will focus on Montgomery incomplete addition here (because only 3 constraints)
- Curve equation: $y_2 = x_3 - 40962.x_2 + x$
- Incomplete addition:
  $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

  $\lambda = (y_2 - y_1)/(x_2 - x_1)$
  $x_3 = \lambda_2 - 40962 - x_1 - x_2$
  $y_3 = (x_1 - x_3).\lambda - y_1$

- As constraints:
  $(x_2 - x_1) \times (\lambda) = (y_2 - y_1)$
  $(\lambda) \times (\lambda) = (x_1 + x_2 + x_3 + 40962)$
  $(x_1 - x_3) \times (\lambda) = (y_1 + y_3)$

- Look how pretty this is: symmetry of the geometric interpretation is preserved in the constraint system, no fluff
- Warning: here be dragons (***incomplete*** addition). But we can tame them.
- Here (at least) is where we need proofs. In the Sapling spec (https://github.com/zcash/zips/blob/master/protocol/sapling.pdf) we prove for example that we can avoid the unhandled addition cases, for points in the prime-order subgroup, by avoiding repeated indices.

Side rant

- 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! <u>DO PROOFS OR YOU WILL FAIL</u>
- 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.
- If you don’t have a proof, at least have an informal argument.
- Do what I say, not what I do (there were/are missing proofs at the time we needed to commit to the Sapling MPC).

Back to elliptic curve stuff:

- Can we reduce cost of addition or doubling further? Or argue for optimality? Other curve shapes?
- Fun, accessible math!
  Add this to pure math syllabuses :-)

Optimization techniques

- 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/mimimize 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.
- Merge to do two things at once
  Example: merging with boolean constraints in constant comparisons.
- Specialize for constants
  Example: lookup from a constant window table in fixed-base scalar mult
- 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 F elements and you can do non-boolean arithmetic on them.
- The most efficient operations are those you can remove.

Poly-F

- Carter-Wegman MAC, like Poly1305, but for F.
- No need for Poly1305 performance hacks.

- Poly1305 is pretty efficient in a circuit, Poly-F is super efficient
- 1 M per F-sized block, plus a cipher (e.g. MiMC 640 M).

The crypto landscape

Protocols

EC-based primitives (hashes, commitments, key exchange)

Scalar multiplication (fixed, variable, multiscalar)

Curve arithmetic

Algebraic primitives (MiMC, Poly-F, ...)

Boring crypto (BLAKE2s, AES, Poly1305, ChaCha20)

Bit-twiddling tricks

Not crypto, but worth optimizing (comparisons, n-ary boolean ops).

What do we know how to do efficiently (and already trust)?
- One-way function (EC scalar mult)
- Key exchange (EC)
- CRH (e.g. Pedersen hash)
- Commitment (e.g. Pedersen commitment)

What could we do efficiently given a cheap PRF or "hash hammer"?
- Useful constructions:
  PRP → PRF (switching lemma)
  PRF → PRP (e.g. Feistel)
  big enough PRP → CRH or hash hammer (fix key and truncate; sponge; other hashing modes)
  PRF + MAC → AEAD
- Signatures (e.g. Schnorr variants need a hash hammer)

Hard but feasible for some applications
- Pairing-based crypto (useful for recursive proof validation)

What can't be done efficiently for now?
- Bignum arithmetic not over F, and public key schemes dependent on it.

Rerandomized signatures
- Basic idea: sign with a randomized private key rsk for pubkey rk.

Publish (sig, rk, proof), where the proof statement is “given rk : PubKey, I know (alpha : Randomizer, ak : PubKey) such that rk is a randomization using alpha of ak (and ak is the right key)”

- The signer can delegate to a prover who doesn’t need the original key ask. The signer must know it because they know rsk and gave the prover alpha, and the randomization is reversible.
- Used in Sapling for spend authorization
  -- e.g. allowing spends to be authorized by a hardware wallet that can’t make (or validate) proofs.
- Signature schemes are specialized zk proofs.
- More generally: use a combination of a zk-SNARK and some kind of special-purpose zk proof.

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 outside 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.