Formal Verification

- Want no bugs: strong guarantee of correctness
- Active research topic

Crypto Code

- Many schemes for computing cryptographic signatures for your messages
  - ECDSA, EdDSA
  - Just math formulas for what a signature is
  - Implementing these signatures is tricky because we need big numbers with modular arithmetic

Big Number Performance

- We represent a large number using multiple "limbs" that are each 64 bit integers
- Naive packed form: number = $x_0 + 2^{64}x_1 + 2^{128}x_2 + \ldots$
  - This leads to problems with propagating the carry bits, which can lead to a timing attack since sometimes the numbers will take longer
- Unpacked: number = $x_0 + 2^{51}x_1 + 2^{102}x_2 + \ldots$
  - There are now multiple canonical representations for a single numbers since we can represent 2^52 in multiple ways
  - We do now need to choose when we will propagate the carry
    - Need to make sure things don't overflow past the 2^64 boundary per limb

Verification Defenses

- Potential problems:
  - Bad scheme:
    - Not handled, this is an issue with the math
  - Buffer overflow:
    - Memory safety
  - Missing correctness proof
    - Functional correctness
    - Code could return the wrong result
  - Timing channel attack
    - Secret independence
    - Want to keep the secret safe from timing attacks

Proof Verifier

- We have a spec that tells us how the function should behave
- We pass this and the implementation code into the verifier, which uses the SMT solver K3 along with some extra proof hints
- If the implementation code is satisfies, we can pass this into a compiler to produce a binary
- Proof: relies on Hoare logic
  - Organized as preconditions and post conditions
  - F* lists these as requires and ensures
- Important tool for writing these pre/post conditions:
  - Abstraction functions
  - Example for squaring a big number
    - Precondition: abs(state) = a
    - Post condition: abs(state') = a' and a' = a^2
- When we use heap things, we also require that passed in pointers must be:

    - Point to a value with memory contents
    - a and b are disjoint
    - Ensure memory safety

Secret Independence

- How to avoid timing attacks
    - Avoiding branching on secrets
    - Avoid divisions with secrets
        - 0s make it easy to do timing attacks
    - No using secrets as a pointer / offsets
- To enforce these with types in F*:
    - uint32_s is just a uint32 in C
    - F* does not offer certain operations for this
        - Does not support comparison operators
        - No divide
        - No pointer arithmetic
    - Supports:
        - eq_mask in constant time
        - Allows you to compare two secrets with each other and its value is either all 1s if a = b and 0 otherwise
        - You can use this as a mask to avoid branching