|galois|

Verified Cryptography for Everybody

Mike Dodds
miked@galois.com

Who am I?

- PhD, York: graph grammars, graph transformation, pointer verification
- Post-doc, Cambridge: separation logic, concurrency, tool building
- Assistant prof, York: (more) separation logic, relaxed memory
- Principal, Galois: crypto, parser security, distributed systems, AI/ML proof repair…

What is Galois anyway?

130+ person industrial lab based in Portland OR, USA

Programming languages research meets real-world applications

Our favorite tools:

- Automated solvers
- Interactive theorem provers
- Safe programming languages
- Fancy type systems

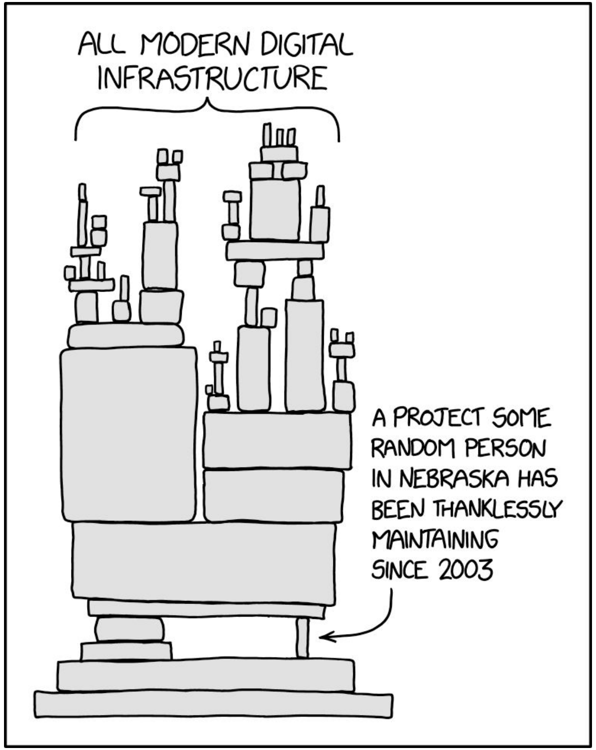
This talk: cryptographic primitives

The building blocks of security:

- Block ciphers: AES, …
- Hash functions: SHA-2, …
- Signature functions: ECDSA, BLS, …

Eg:

- Core libraries: OpenSSL, BoringSSL, …
- Exotic stuff: quantum-resistant primitives, blockchain-specific libraries



Source: https://xkcd.com/2347/ - CC BY-NC 2.5

Who cares?

Cryptographic libraries matter:

- *(billions of users) * (millions of calls per day)*
- Security-critical in nearly every dimension
- Highly optimized, incredibly gnarly code, very difficult to audit

But also:

- A small number of libraries cover nearly all usage
- The code is highly encapsulated and changes very slowly

Verifying *this* code ⇒ verified cryptographic code for everybody

Galois does difficult proofs

- 2018: "*Continuous Formal Verification of Amazon s2n*" (CAV)
  - Target: core components of Amazon's TLS library
- 2021: "*Verified Cryptographic Code for Everybody*" (CAV)
  - Target: core components of AWS-LibCrypto (OpenSSL fork)
- 2022: Verification of the `blst` library
  - Target: signature library focused on performance and security

What are we verifying, anyway?

Threat model for cryptographic primitives

Code crashes

- More precisely, C / LLVM *undefined behaviour,* e.g. writes out of memory bounds
- *Potential attack:* break memory safety / security on host

Code does not not implement the algorithm correctly

- Eg. might not compute AES-GCM correctly for some input
- *Potential attack:* decrypt messages in transit

Out of scope:

- Side-channels generally
  - Timing - eg. different messages take different times to decrypt
  - Microarchitectural - eg SPECTRE / MELTDOWN etc

- Algorithm-level cryptographic security properties
  - We verify: *code implements algorithm*
  - Not verified: *algorithm is cryptographically secure*

Verification means program equivalence

complex program $\approx$ simple program

⇧

That is, a *reference implementation* or *specification*

So, verification is just fancy testing

Testing:

$$\text{program}(\text{input}) \neq \text{crash} \quad \land \quad \text{program}(\text{input}) \approx \text{expected\_result}$$

Formal verification:

$$\forall\ \text{input}.$$

$$\text{program}(\text{input}) \neq \text{crash} \quad \land \quad \text{program}(\text{input}) \approx \text{specification}(\text{input})$$

E.g hash-based message authentication code

BoringSSL HMAC code $\approx$ 'book' HMAC (RFC 2104)

E.g hash-based message authentication code:

```
#include <openssl/hmac.h>

#include <assert.h>
#include <string.h>

#include <openssl/digest.h>
#include <openssl/mem.h>

#include "../../internal.h"
#include "../service_indicator/internal.h"

typedef int (*HashInit)(void *);
typedef int (*HashUpdate)(void *, const void*, size_t);
typedef int (*HashFinal)(uint8_t *, void*);

struct hmac_methods_st {
  const EVP_MD* evp_md;
  HashInit init;
  HashUpdate update;
  HashFinal finalize; // Not named final to avoid keywords
};

// We need trampolines from the generic void* methods we use to the properly typed un
// Without these methods some control flow integrity checks will fail because the fun
// do not exactly match the destination functions. (Namely function pointers use void
// while the destination functions have specific pointer types for the relevant conte
//
// This also includes hash-specific static assertions as they can be added.
#define MD_TRAMPOLINES_EXPLICIT(HASH_NAME, HASH_CTX, HASH_CBLOCK)           \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Init(void *);                  \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Update(void *, const void *,   \
                                                    size_t);                \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Final(uint8_t *, void *);      \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Init(void *ctx) {              \
    return HASH_NAME##_Init((HASH_CTX *)ctx);                               \
```

https://github.com/awslabs/aws-lc/blob/main/crypto/fipsmodule/hmac/hmac.c

≈

‘book’ HMAC (RFC 2104)

E.g hash-based message authentication code:

```
#include <openssl/hmac.h>

#include <assert.h>
#include <string.h>

#include <openssl/digest.h>
#include <openssl/mem.h>

#include "../../internal.h"
#include "../service_indicator/internal.h"

typedef int (*HashInit)(void *);
typedef int (*HashUpdate)(void *, const void*, size_t);
typedef int (*HashFinal)(uint8_t *, void*);

struct hmac_methods_st {
  const EVP_MD* evp_md;
  HashInit init;
  HashUpdate update;
  HashFinal finalize; // Not named final to avoid keywords
};

// We need trampolines from the generic void* methods we use to the properly typed un
// Without these methods some control flow integrity checks will fail because the fun
// do not exactly match the destination functions. (Namely function pointers use void
// while the destination functions have specific pointer types for the relevant conte
//
// This also includes hash-specific static assertions as they can be added.
#define MD_TRAMPOLINES_EXPLICIT(HASH_NAME, HASH_CTX, HASH_CBLOCK)              \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Init(void *);                     \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Update(void *, const void *,      \
                                                     size_t);                   \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Final(uint8_t *, void *);         \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Init(void *ctx) {                 \
    return HASH_NAME##_Init((HASH_CTX *)ctx);                                  \
```

https://github.com/awslabs/aws-lc/blob/main/crypto/fipsmodule/hmac/hmac.c

≈

```
We define two fixed and different strings ipad and opad as follows
(the 'i' and 'o' are mnemonics for inner and outer):

                  ipad = the byte 0x36 repeated B times
                  opad = the byte 0x5C repeated B times.

To compute HMAC over the data `text' we perform

                    H(K XOR opad, H(K XOR ipad, text))
```

https://datatracker.ietf.org/doc/html/rfc2104.html

We can't verify a specification in natural language, like RFC2104

Solution: Convert natural language RFC into a high-level specification language, *Cryptol* - https://cryptol.net/

The specification is close enough for cryptographers to audit and establish high confidence

```
We define two fixed and different strings ipad and opad as follows
(the 'i' and 'o' are mnemonics for inner and outer):

                ipad = the byte 0x36 repeated B times
                opad = the byte 0x5C repeated B times.

To compute HMAC over the data `text' we perform

                  H(K XOR opad, H(K XOR ipad, text))
```

https://datatracker.ietf.org/doc/html/rfc2104.html

⇒

```
hmac hash hash2 hash3 key message = hash2 (okey # internal)
 where
  ks = kinit hash3 key // K'
  okey = [k ^ 0x5C | k <- ks] // K' xor opad
  ikey = [k ^ 0x36 | k <- ks] // K' xor ipad
  // H((K' xor ipad) || message)
  internal = split (hash (ikey # message))
```

https://github.com/GaloisInc/cryptol-specs/blob/master/Primitive/Symmetric/MAC/HMAC.cry

E.g hash-based message authentication code:

BoringSSL HMAC code

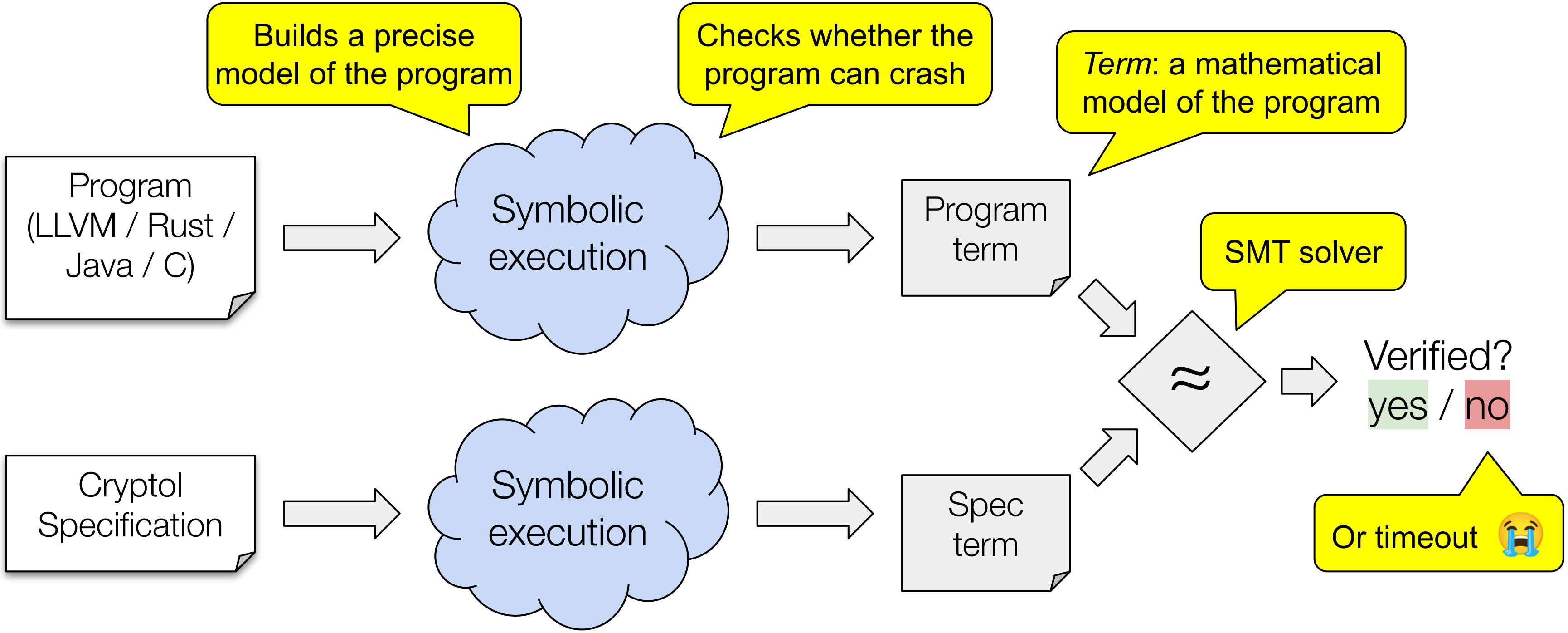
```c
#include <openssl/hmac.h>

#include <assert.h>
#include <string.h>

#include <openssl/digest.h>
#include <openssl/mem.h>

#include "../../internal.h"
#include "../service_indicator/internal.h"

typedef int (*HashInit)(void *);
typedef int (*HashUpdate)(void *, const void*, size_t);
typedef int (*HashFinal)(uint8_t *, void*);

struct hmac_methods_st {
  const EVP_MD* evp_md;
  HashInit init;
  HashUpdate update;
  HashFinal finalize; // Not named final to avoid keywords
};

// We need trampolines from the generic void* methods we use to the properly typed un
// Without these methods some control flow integrity checks will fail because the fun
// do not exactly match the destination functions. (Namely function pointers use voic
// while the destination functions have specific pointer types for the relevant conte
//
// This also includes hash-specific static assertions as they can be added.
#define MD_TRAMPOLINES_EXPLICIT(HASH_NAME, HASH_CTX, HASH_CBLOCK)           \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Init(void *);                  \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Update(void *, const void *,   \
                                                    size_t);                \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Final(uint8_t *, void *);      \
  static int AWS_LC_TRAMPOLINE_##HASH_NAME##_Init(void *ctx) {              \
    return HASH_NAME##_Init((HASH_CTX *)ctx);                               \
```

≈

Cryptol HMAC specification

```
hmac hash hash2 hash3 key message = hash2 (okey # internal)
 where
  ks = kinit hash3 key // K’
  okey = [k ^ 0x5C | k <- ks] // K’ xor opad
  ikey = [k ^ 0x36 | k <- ks] // K’ xor ipad
  // H((K’ xor ipad) || message)
  internal = split (hash (ikey # message))
```

Proof tool: *SAW* (Software Analysis Workbench)



Result - high confidence of this:

$\forall$ input.

$$\text{primitive}(\text{input}) \neq \text{crash} \quad \wedge \quad \text{primitive}(\text{input}) \approx \text{cryptol\_spec}(\text{input})$$

Verified for AWS-libcrypto:

- HMAC with SHA-384
- SHA-2 384 & 512
- AES-GCM 256
- AES-KW(P) 256
- ECDSA with P-384, SHA-384
- ECDH with P-384

Verified for s2n TLS library

- DRBG
- HMAC
- TLS 1.2 state machine

Verified for blst:

- All operations

Difficult proofs are difficult (for now)

Verifying cryptography is easy!

- Code is mostly bounded in input size; loops can be unrolled
- Data-structures are static; v. restricted pointers / dynamic allocation
- Interfaces are fixed and have precise, commonly agreed specifications (RFCs / white papers)
- Code is extremely stable over time; major libraries share a lot of code

Conservation of difficulty rule:

Software that was difficult to write will be difficult to verify

Corollary: *software that was easy to write is easy to verify!*

NB: this is a joke

Why is it difficult to verify cryptography?

'book' HMAC ≈ HMAC1 ≈ HMAC2 ≈ . . . ≈ OpenSSL HMAC

⇧

- Multiple use cases / platforms
- Legacy considerations
- (most important) *optimization*

Intuition: each step requires a theorem

Verifying cryptography is difficult

- Generally: some of the most heavily optimized code in existence
- Implementations use C and specialized x86 instructions. Some of the code is generated by Perl scripts
- Many optimizations rely on facts about math(s) in order to be sound
- Many optimizations break abstraction boundaries, e.g. by pipelining instructions, unrolling loops

Tools for controlling difficulty in SAW

- ~~Change the code~~
- Rewriting / uninterpreted functions
- Composition

Changing the code is incredibly powerful!

Very similar programs can have dramatically different verification characteristics

Why? Some hypotheses:

- Many obvious-to-humans equivalences depend on deep theorems
- Solver nondeterminism - small perturbations can make a goal unsolvable
- Problem diversity - any tool makes some patterns easier and other patterns harder

We (mostly) don't change the code

Built-for-verification systems are awesome (seL4, HACL*, …)

But: *we want to verify the code everybody is using*

Engineers might trust pre-existing code more than verified alternatives:

- Existing code has been tested / fuzzed / inspected
- Existing code has been used for 1000s of hours in production
- Existing code may be certified, e.g. through FIPS (HUGE deal)
- Built-for-verification code may not have a long-term support story

Tools for controlling difficulty in SAW

- ~~Change the code~~
- Rewriting / uninterpreted functions
- Composition

Rewriting / uninterpreted functions

Spec Term

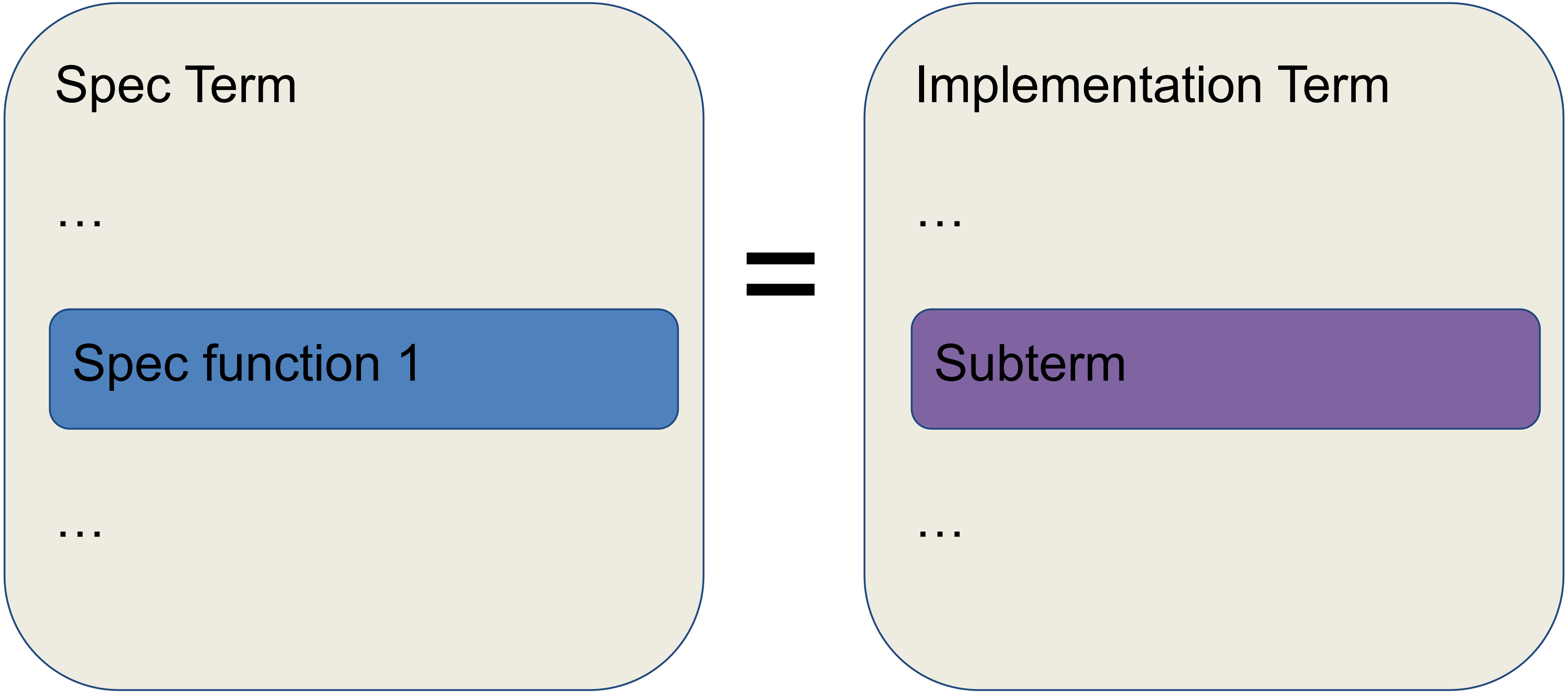
=

Implementation Term

Do a sub-proof

Spec function 1 = Subterm

Rewrite



Rewrite

Spec Term

…

Spec function 1

…

=

Implementation Term

…

Spec function 1

…

Uninterpret and prove the overall equivalence

Spec Term

…

Spec function 1

…

=

Implementation Term

…

Spec function 1

…

Rewrites in practice (from SHA 384)

Cryptol

```
S0 x = (x >>> 28) ^ (x >>> 34) ^ (x >>> 39)
```

Perl that generates assembly

```
'&ror           ($a1,39-34)',
'&xor           ($a1,$a)',
'&ror           ($a1,34-28)',
'&xor           ($a1,$a)',
'&ror           ($a1,28)',
```


Rewrites in practice (from SHA 384)

Cryptol

```
S0 x = (x >>> 28) ^ (x >>> 34) ^ (x >>> 39)
```

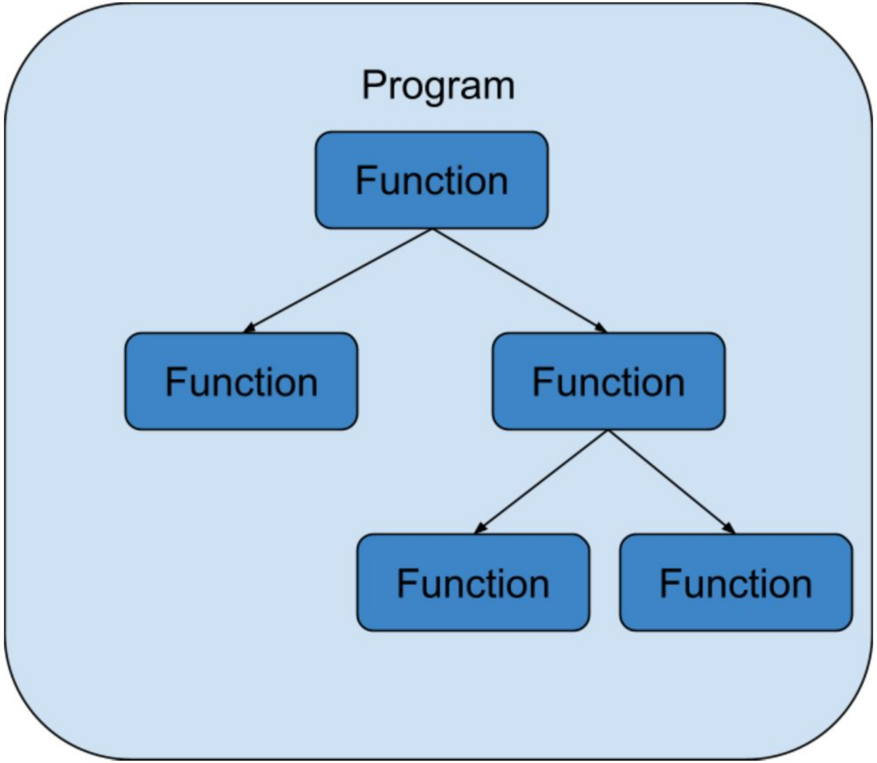
SAW Rewrite Rule

```
Sigma0_thm <- prove_folding_theorem
                {{ \x -> (x ^ ((x ^ (x <<< 59)) <<< 58)) <<< 36 == S0 x }};
```

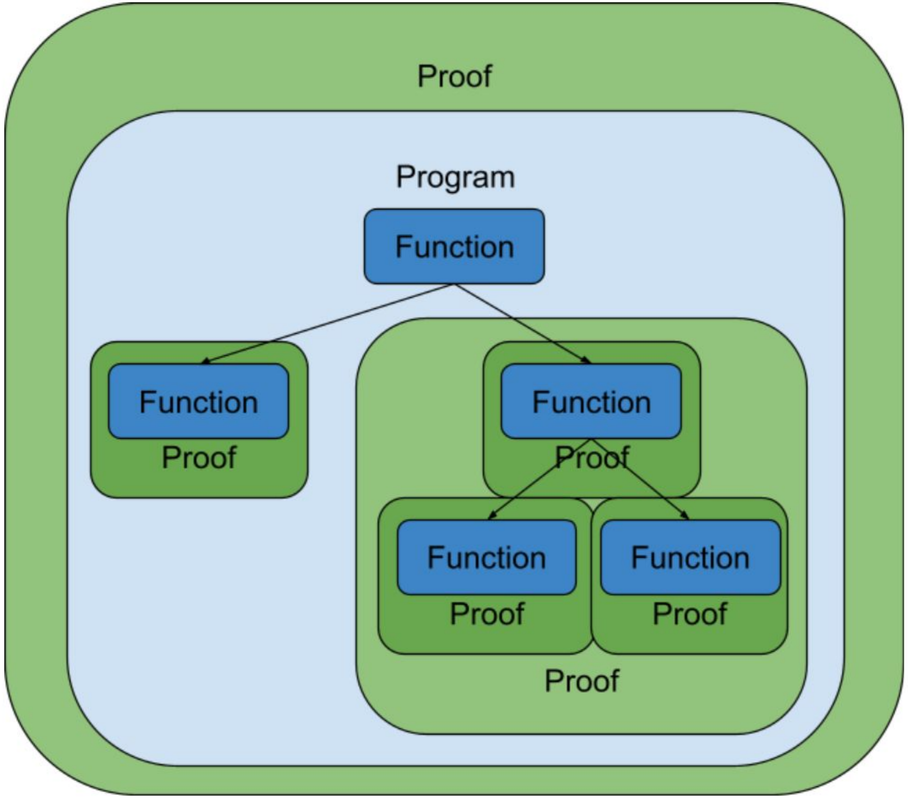

Tools for controlling difficulty in SAW

- ~~Change the code~~
- Rewriting / uninterpreted functions
- Composition

Programs come with structure



Break the proof down with that structure



Composition in SAW

- During symbolic execution a called function can be replaced by its specification
- Saves symbolic execution time
- Can result in simpler formulas

Composition in SAW

```
let main : TopLevel () = do {
    m      <- llvm_load_module "salsa20.bc";
    qr     <- crucible_llvm_verify m "s20_quarterround" []      false quarterround_setup   z3;
    rr     <- crucible_llvm_verify m "s20_rowround"     [qr]    false rowround_setup       z3;
    cr     <- crucible_llvm_verify m "s20_columnround"  [qr]    false columnround_setup    z3;
    dr     <- crucible_llvm_verify m "s20_doubleround"  [cr,rr] false doubleround_setup    z3;
    s20    <- crucible_llvm_verify m "s20_hash"         [dr]    false salsa20_setup        z3;
    s20e32 <- crucible_llvm_verify m "s20_expand32"     [s20]   true  salsa20_expansion_32 z3;
    s20encrypt_63 <- crucible_llvm_verify m "s20_crypt32" [s20e32] true (s20_encrypt32 63) z3;
    s20encrypt_64 <- crucible_llvm_verify m "s20_crypt32" [s20e32] true (s20_encrypt32 64) z3;
    s20encrypt_65 <- crucible_llvm_verify m "s20_crypt32" [s20e32] true (s20_encrypt32 65) z3;

    print "Done!";
};
```

Composition in SAW

C function name — Overrides — Path satisfiability checking — SAW Specification — Tactic

```
let main : TopLevel () = do {
    m       <- llvm_load_module "salsa20.bc";
    qr      <- crucible_llvm_verify m "s20_quarterround" []      false quarterround_setup   z3;
    rr      <- crucible_llvm_verify m "s20_rowround"     [qr]    false rowround_setup       z3;
    cr      <- crucible_llvm_verify m "s20_columnround"  [qr]    false columnround_setup    z3;
    dr      <- crucible_llvm_verify m "s20_doubleround"  [cr,rr] false doubleround_setup    z3;
    s20     <- crucible_llvm_verify m "s20_hash"         [dr]    false salsa20_setup        z3;
    s20e32 <- crucible_llvm_verify m "s20_expand32"      [s20]   true  salsa20_expansion_32 z3;
    s20encrypt_63 <- crucible_llvm_verify m "s20_crypt32" [s20e32] true (s20_encrypt32 63) z3;
    s20encrypt_64 <- crucible_llvm_verify m "s20_crypt32" [s20e32] true (s20_encrypt32 64) z3;
    s20encrypt_65 <- crucible_llvm_verify m "s20_crypt32" [s20e32] true (s20_encrypt32 65) z3;

    print "Done!";
};
```

Composition has a cost

- If we have to specify internal functions, we have to specify their state
- We might need internal specs too specific for general use
- We might need multiple specs for the same function

Composition has a cost

Example, monolithic cryptography functions

```
monolithic : key -> message -> output
```

Vs iterative:

```
init : key -> state
update : state -> message -> state
final : state -> output

monolithic k m = final (update (init key) message)
```

Proofs as Engineering Tools

Problems engineers care about:

- Increasing confidence in particularly critical functionality
- Catching important bugs or eliminating classes of bugs
- Increasing test coverage
- Passing certification more quickly / cheaply
- Making justified claims about reliability / security to customers

Proofs can help with these problems!

Proofs are one expensive tool in the reliability toolbox

along with testing, fuzzing code review, safe languages, CI/CD, good dev practices, hiring clever people, using well-tested components …

Cost/benefit matters - *at the current margin*

Unrealistic: "*Let's formally verify our whole stack*"

Realistic: "*Should we spend $X and Y months on this particular proof - or spend the same budget on tests / fuzzing?*"

Proofs have to win the argument project-by-project

- low, predictable costs
- large, quantifiable benefits

Predictability matters

How much will a given proof cost, in $$ / time / expertise?

Current proof projects are unpredictable at multiple scales:

- Micro: will the solver discharge a particular goal?
- Macro: how difficult is a particular piece of software to verify?

We mitigate this problem with team experience across multiple proofs and careful proof design

More R&D needed

Target selection matters

Good qualities:

- Security / safety critical; faults would be disastrous
- A large number of users rely on the code
- Well-understood interfaces that can be phrased in math
- Stable, slow-changing codebase
- Limited use of ‘difficult’ features: memory allocation, complex invariants, embedded assembly …

Not all of these qualities are necessary, and tools are developing rapidly

Understandability matters

Who are the users for the proof? What benefit do they gain from proof?

Multiple audiences:

- A customer for a product that has been improved
- An engineer who will interact with the proof
- Formal methods experts

Each audience needs an explanation that is *understandable* and *accurate*

More R&D needed

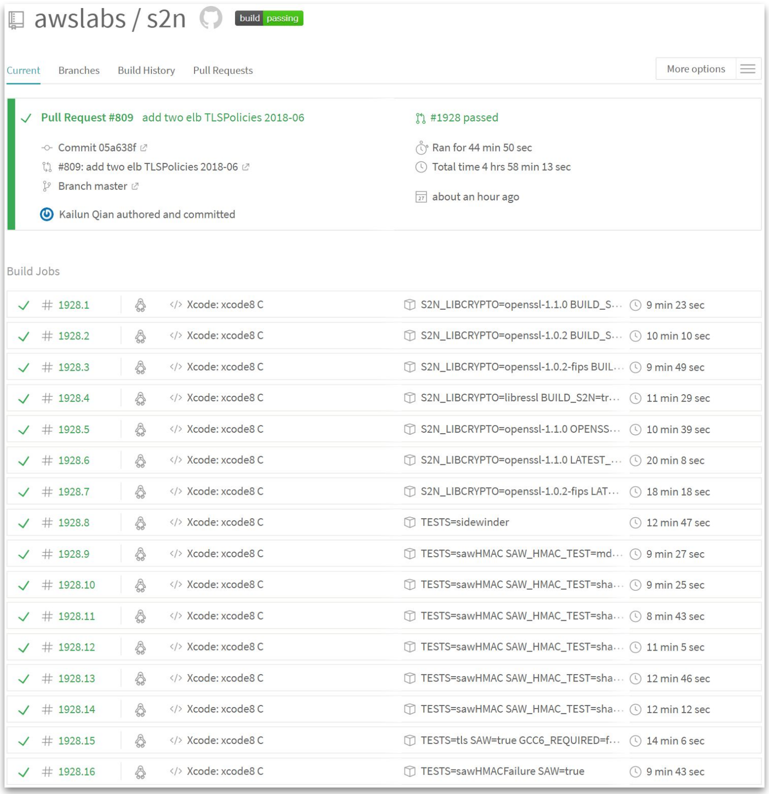
Integration matters

Proofs have to fit into the existing engineering workflow (lowers proof cost)

Our proofs run in CI/CD on every commit to the codebase

This is often a significant challenge:

- Proofs have to run within time / memory budgets
- Proofs block deployment - need to fix problems quickly



Proof engineering matters

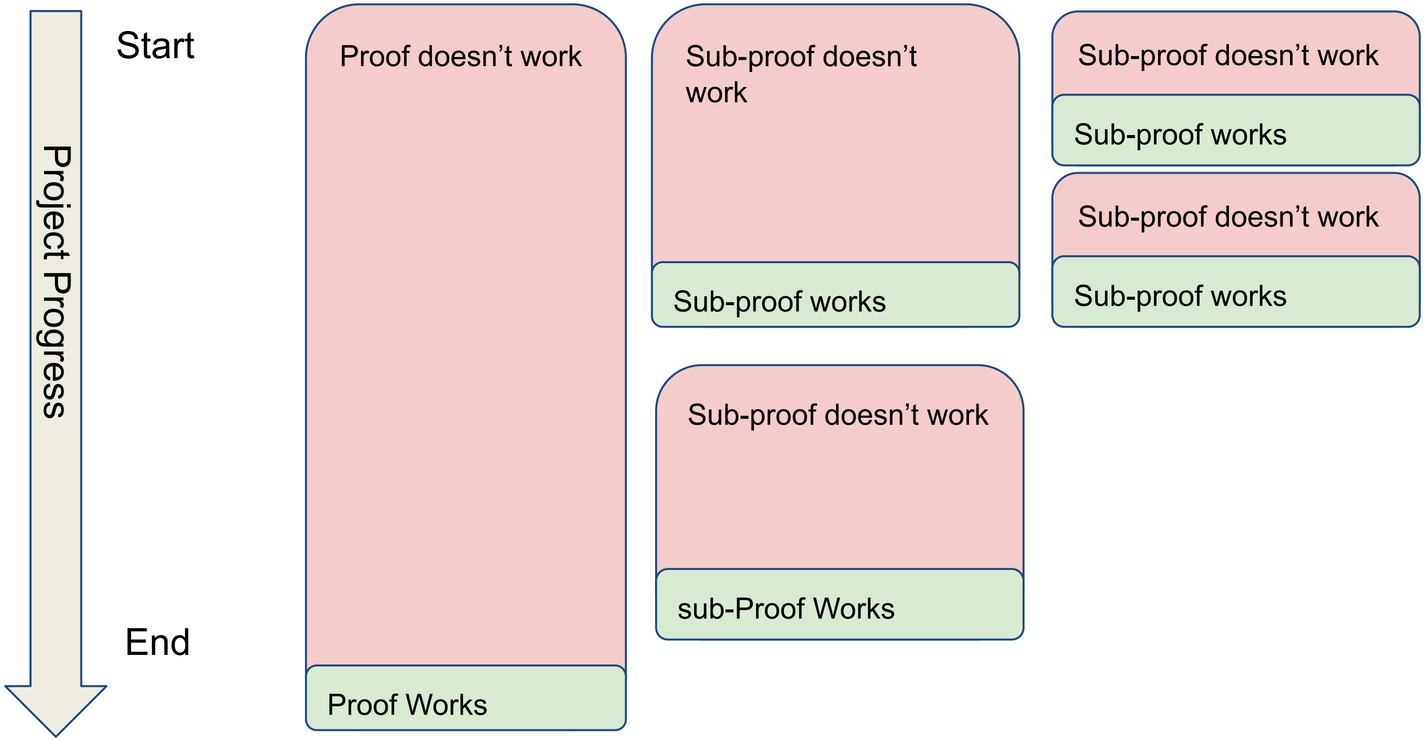
Earlier: *“Software that was difficult to write will be difficult to verify”*

Building cryptographic libraries required:

- Well-designed mature tools
- A professional, experienced team of engineers
- Well-tested engineering practices
- A significant amount of time

Formally verifying cryptographic libraries requires the same!

How proof engineers spend their time



We spend most of our time with broken proofs

If we didn't the effort would be done!

UX for broken proofs is the most important UX for proof tools

- Solver feedback
- Execution exploration
- Goal exploration and manipulation

More R&D needed!

Some questions when deploying proofs:

- *Who cares?* Who will have their problem solved by the proof?
- What will the proof cost and how long will it take?
- What threats will the proof prevent? How are they currently prevented? What do they currently cost?
- How will the proof fit into the existing engineering process?
- Who will build and maintain the proof?
- What happens when the system changes?

These are mostly just rephrased Heilmeier catechisms:
https://www.darpa.mil/work-with-us/heilmeier-catechism

Wrap-up: Verified Cryptography for Everybody

Wrap-up

- Galois has verified cryptographic libraries used by everybody
- Proofs are equivalences between executable specifications (written in Cryptol) and implementation code (C / x86)
- Proofs are difficult thanks to extremely gnarly optimisations; we control difficulty using composition and rewriting (& experience)
- Proofs are tools for engineers; cost/benefit tradeoffs matter in multiple dimensions

Further reading

- *Verified Cryptographic Code for Everybody* (CAV 2021)
  Technical paper on the AWS-LC proof and tools.
  https://doi.org/10.1007/978-3-030-81685-8_31

- *Formally Verifying Industry Cryptography* (S&P 2022)
  Non-technical paper on our proof engineering process.
  https://doi.org/10.1109/MSEC.2022.3153035


galois