LA-UR-22-26855

Approved for public release; distribution is unlimited.

Title:	Secure System Composition and Type Checking using Cryptographic Proofs
Author(s):	Barrack, Daniel Abraham
Intended for:	Presentation to be given to zkSNARK and formal methods researchers, as well as other interested viewers
Issued:	2022-07-13

Secure System Composition and Type Checking using Cryptographic Proofs

Dani Barrack
A-4: Advanced Research in Cyber Systems
Email: dbarrack@lanl.gov

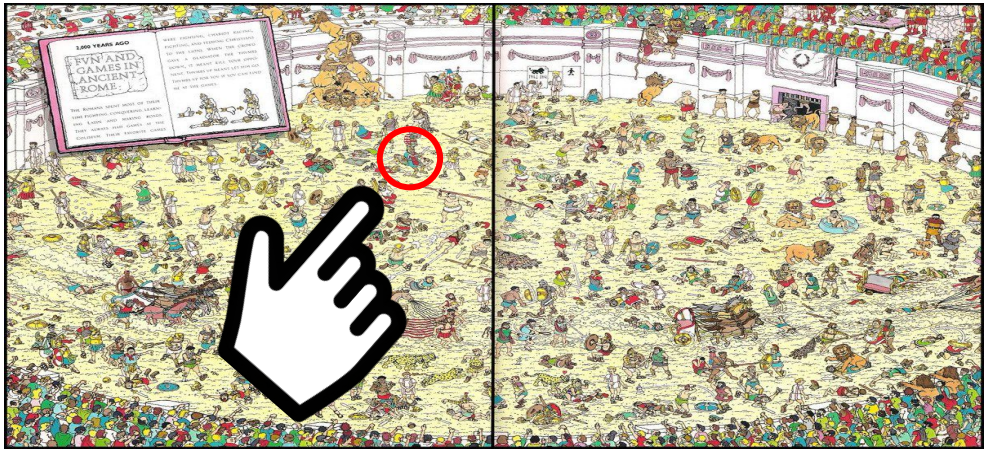
Collaborator: Zachary DeStefano
Mentor: Michael J. Dixon
Co-Mentor: Boris Gelfand

Cryptographic proofs

Zero-Knowledge Proof for *Where's Waldo?*

Example. You want to prove that you have beaten *Where's Waldo?*

- **Traditional Proof:** Point to Waldo to demonstrate you know where he is



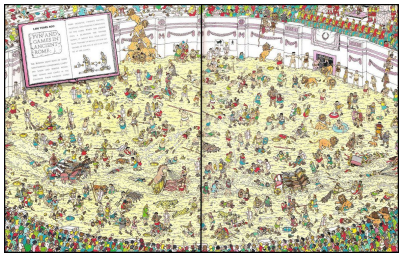
- Not zero-knowledge!

  This kind of proof leaks all information about his location, much more than simply that you have *knowledge* of the location

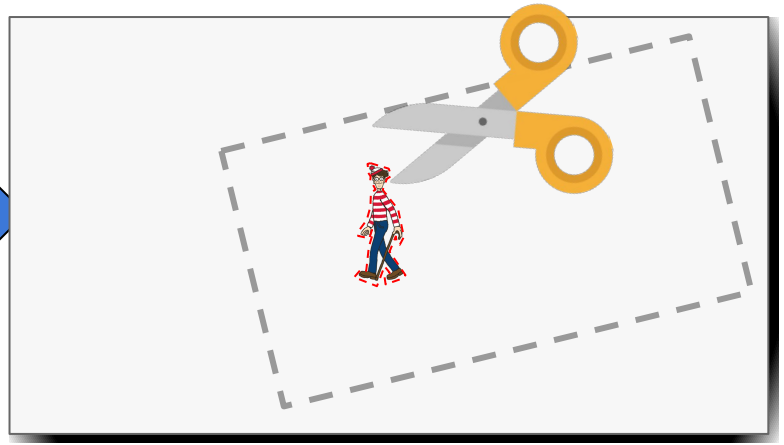
Zero-Knowledge Proof for *Where's Waldo?*

Zero-knowledge proof for "Where's Waldo?"

1. Cut out a Waldo shaped hole in a much larger piece of paper
2. Position the hole over Waldo's location



Slide under paper



This precisely obfuscates Waldo's location while demonstrating knowledge of his whereabouts!

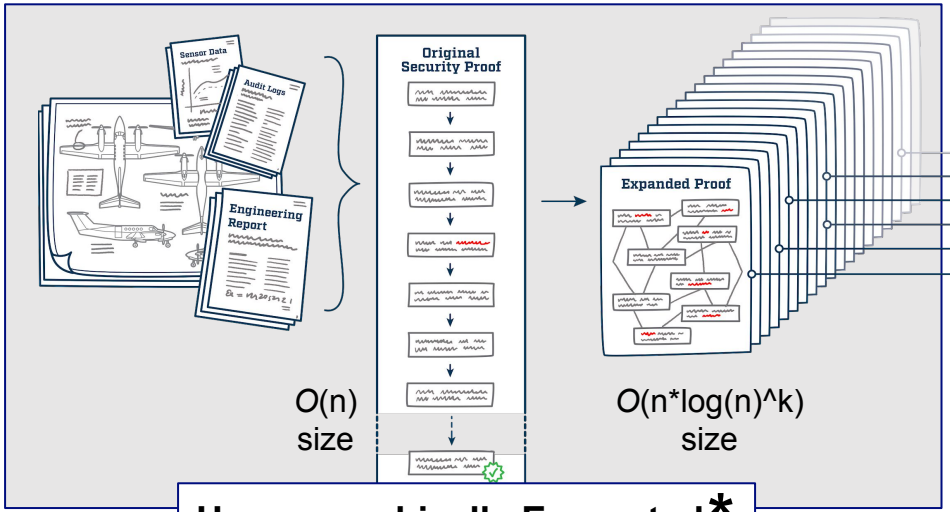
To adversaries, the book underneath could hypothetically be in any random orientation

Zero-Knowledge Proofs and Verifiable Computation

Zero-knowledge proofs (ZKPs) allow us to prove that a claim **IS** true without revealing **WHY** it is true, even if the prover is untrusted and malicious.

zkSNARKs are special ZKPs that are ***tiny*** and ***non-interactive***

Inputs:
Logs
Schematics
Program Traces
Signals
Encryption Keys
Attestations
etc.



Outputs:
Results
&
ZK Proof of Computational Integrity

***with tweaks**

zkSNARK Construction for Program Verification [BCGTV13]

```
int myFunction(int a) {
     int b=a*a-4;
     return 3*b+a;
}
```

Rank-1 Constraint System (R1CS):

$$S \cdot A * S \cdot B = S \cdot C$$

1	0	1	0	1	0
a	1	a	1	a	1
t0	0	t0	0	t0	0
b	0	b	0	b	0

libsnark

- Computation
- Arithmetic Circuit
- R1CS

Proof Representation Of Program Execution

- QAP — Zero Knowledge Added
- LPCP — Succinctness Added
- LIP — Interactivity Removed
- zkSNARK

libsnark backend

Verifier Binary

Prover Binary

zkSNARK for Program Integrity

Cryptographic Proof Systems

Cryptographic proof systems have variable completeness and soundness (slightly distinct from their meaning in formal systems). For non-interactive zero-knowledge proofs we care about:

(Completeness) ℙ[true statement AND verifier accepts] = 1
"Every valid proof will be accepted by a verifier "

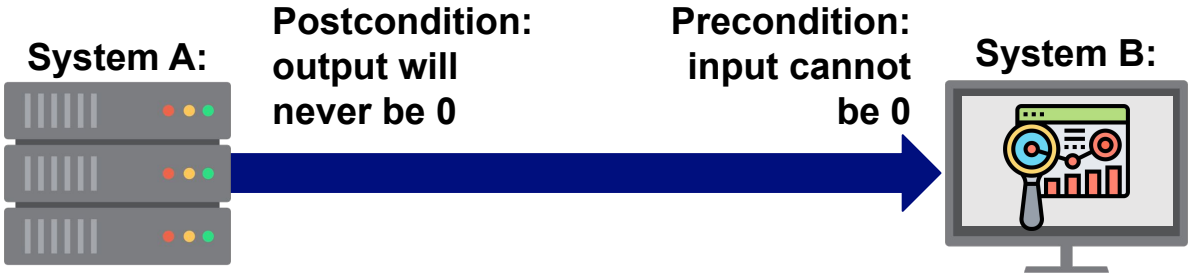
(Soundness) ℙ[false statement AND verifier rejects] = 1 - ε
"Low chance that a proof of a false statement is encountered"

We sacrifice minimal amount of soundness (have to break crypto to produce counter-example) in order to get valuable proof properties

Assuring Safe System Composition

Motivation:

Systems have specific conditions under which they operate correctly. Often these involve restrictions on the data supplied to them.



Type checking is a form of composition checking

Type checking is a kind of specification that defines what kind of parameters a function takes, and what it returns.

Strong static typing gives strong static guarantees

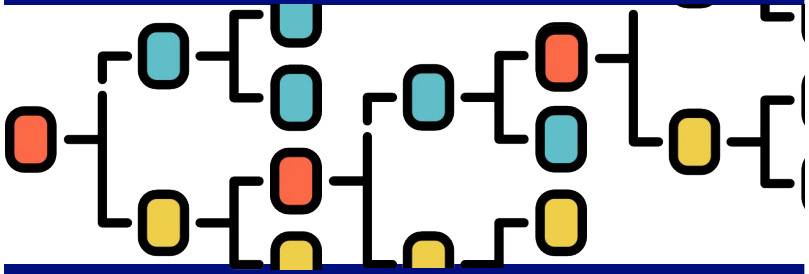
- If it type checks, the program will not result in type errors when run
- Builds proof of correctness internally

```
oneFloor:: Integer -> NonZero Integer
oneFloor x = toNonZero $ max 1 x
```

```
div :: Integer -> NonZero Integer -> Integer
div num denom = num / denom
```

```
floorDiv :: Integer -> Integer -> Integer
floorDiv num denom = div num $ oneFloor denom
```

Challenge: Formally Verifying System Composition

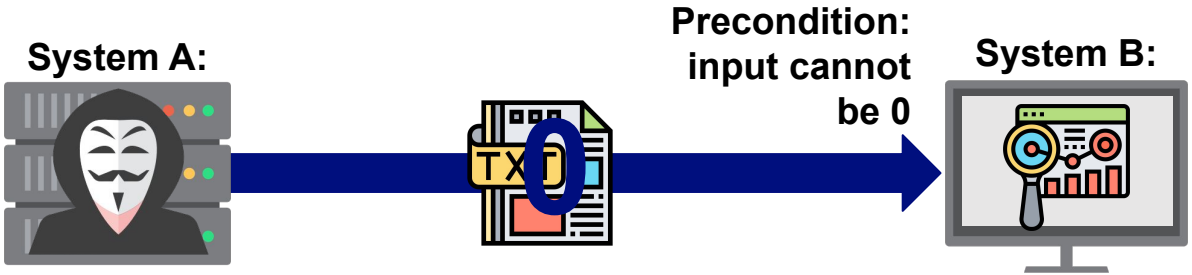


Entire systems can be modeled and verified together, but the state space can become intractable, and this approach assumes trusted computing capabilities.

Individual systems are verified with pre and post-conditions, which are checked at runtime. This may require re-execution of computation or the revealing of sensitive data.

Problem: Guarantees can be lost across systems

In type checking on a single-system program, it is generally assumed that data is not altered in ways not captured by the type-system. One does not generally need to worry about data-integrity of the memory of a running program on a trusted system



Even if on paper two systems have pre and post conditions that are compatible, this implies that the sending system can be trusted to abide by those requirements.

Problem: Not all requirements can be checked easily

Obviously a requirement such as that the input value is not a zero, or a list is non-empty is an easy thing to verify at runtime.

However, facts about what procedures were used to generate the data, data quality as measured by non-public metrics, or relations between data that is confidential may not be detectable without supplying additional privileged information, or re-executing computations.

Difficult Preconditions and Postconditions

```
record QualityMLModel : Set where
 field
   w        : Weights
   errLimit : (error w) < (0.05)
   log      : AuditLog
   logProof : execute log ≡ w
```

```
trainModel : List Inputs → Maybe QualityMLModel
```

```
classifyPoint : Input → QualityMLModel → Class
```

We can generate proofs (or an audit log) of desired properties (e.g. functional correctness) and include them with input to other functions.

This allows us to use a dependent type to assure that only models that were actually trained on actual data and are within a certain error threshold can be used by a classifier.

The audit log and its proof would be **huge**. Instead, we can use a super small zkSNARK to prove this dependent type and pass it along instead. We only need to handle the case where the check fails.

Dependent Types: Sigma types (Dependent pairs)

Sigma types can be thought of as existential statements.

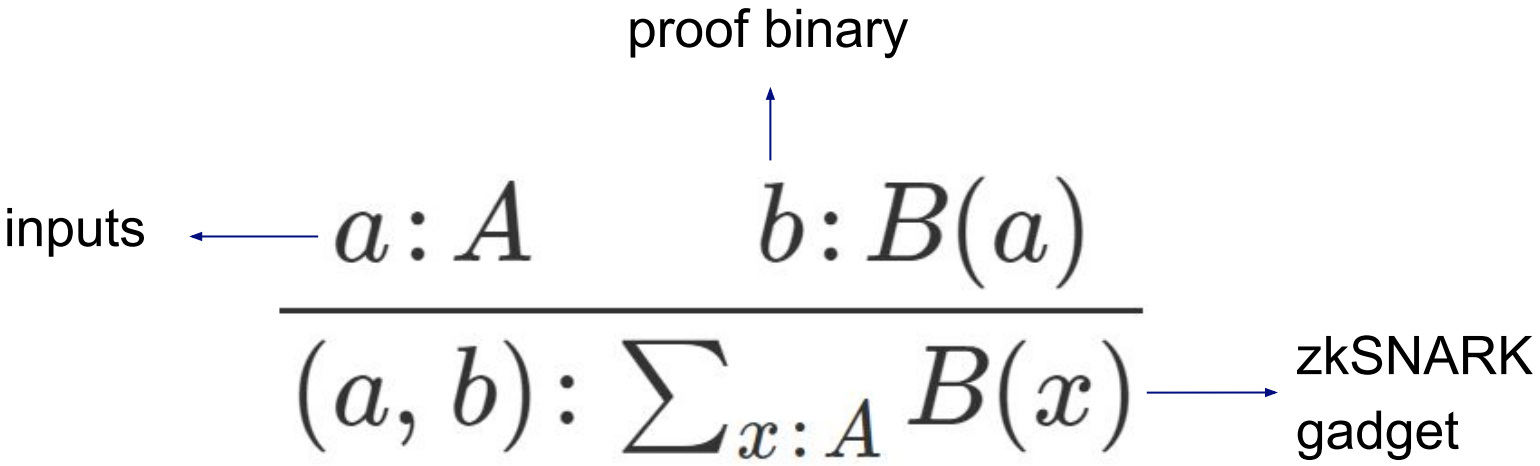
$$\frac{a : A \qquad b : B(a)}{(a, b) : \sum_{x : A} B(x)}$$

The second member of a pair supplies a member of *B*(*x*), when the first member is some x : A such that *B*(*x*) is inhabited.

“There exists some value x that meets these constraints B(x) imposes”

zkSNARKs: A tailored proof of knowledge

zkSNARKs correspond to sigma types.



The inputs correspond to the first member of the pair, the gadget defines the predicate, and the second member of the pair is the proof that the predicate holds over those inputs.

Preconditions and Postconditions with Types

```
record QualityMLModel : Set where
 field
   w         : Weights
   proof     :
    IO (ZKP (∃[ ( w' , inputs ) ]
               ((error w' ) < 0.05 ,
               (train inputs ≡ w')))
```

```
trainModel : List Inputs → Maybe QualityMLModel
```

```
classifyPoint : Input → QualityMLModel → IO Class
```

We can generate proofs (or an audit log) of desired properties (e.g. functional correctness) and include them with input to other functions.

This allows us to use a dependent type to assure that only models that were actually trained on actual data and are within a certain error threshold can be used by a classifier.

The audit log and its proof would be **huge**. Instead, we can use a super small zkSNARK to prove this dependent type and pass it along instead. We only need to handle the case where the check fails.

Formal analysis of programs with ZKP

```
record QualityMLModel : Set where
 field
   w         : Weights
   proof     :
    IO (ZKP (∃[ ( w' , inputs ) ]
               ((error w' ) < 0.05 ,
               (train inputs ≡ w')))
```

```
trainModel : List Inputs → Maybe QualityMLModel
```

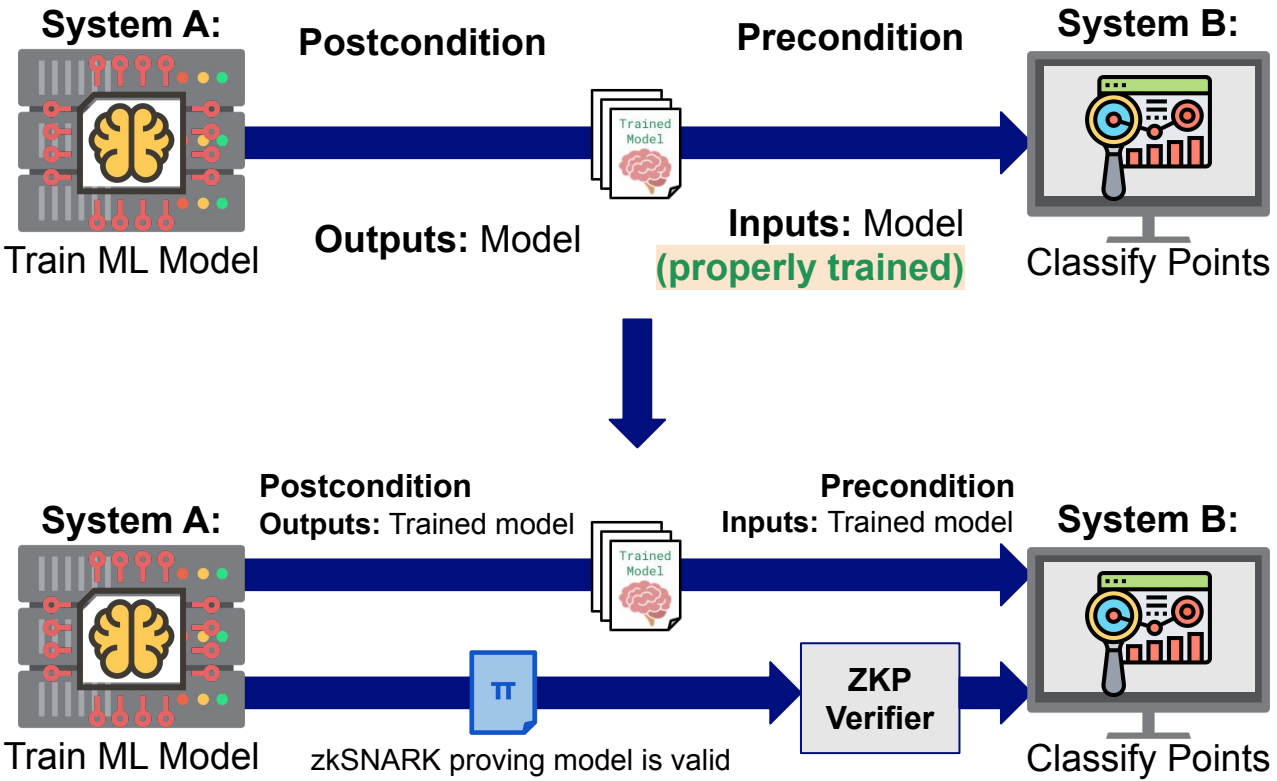
```
classifyPoint : Input → QualityMLModel → IO Class
```

If the implementation of the zkSNARK gadget g accurately captures the predicate of the sigma type, we can postulate that if verification of the zkp succeeds, then the existential it corresponds to is inhabited.

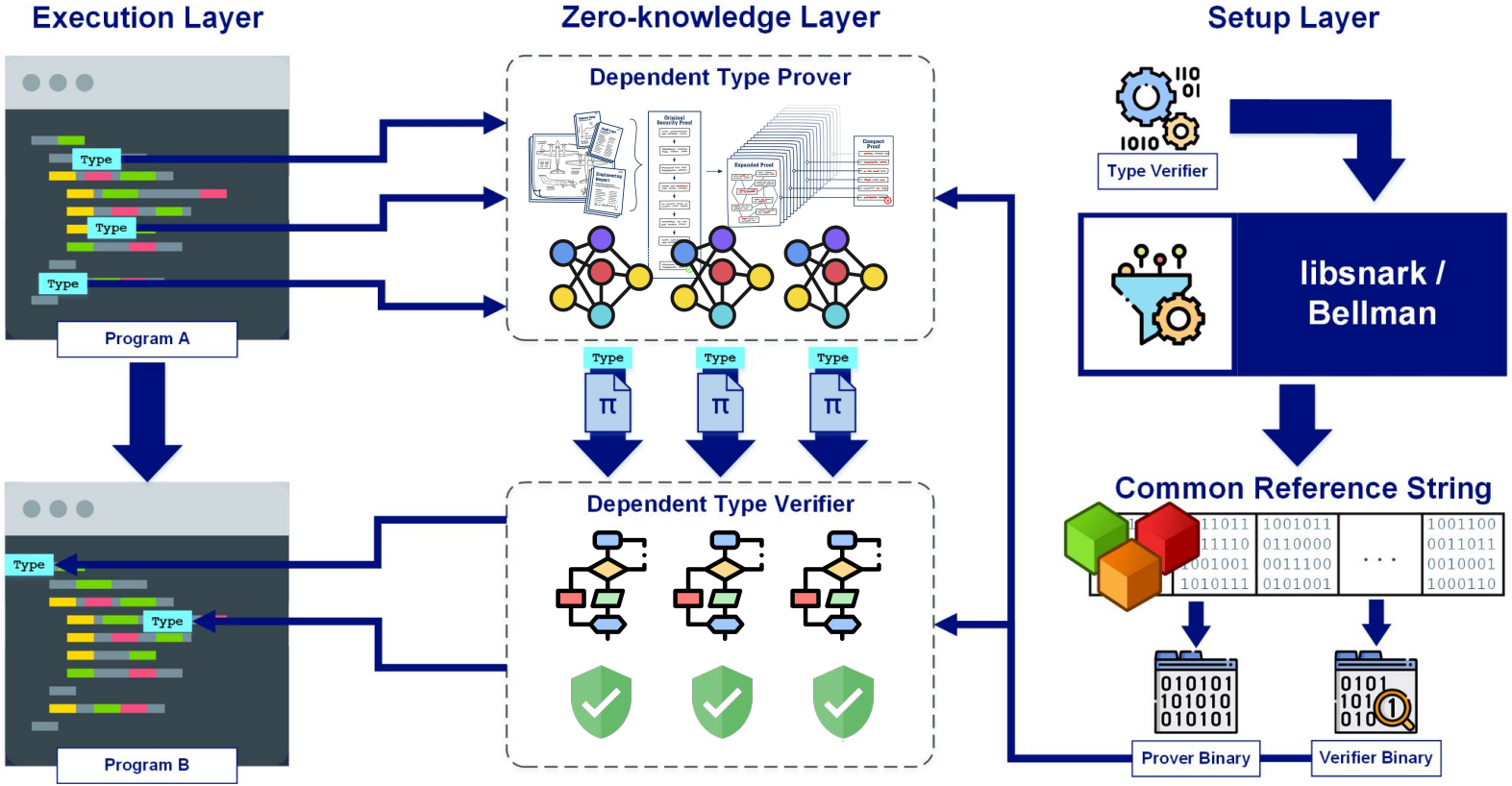
verifyZKP g inputs key = 1 → ∃[inputs] P(inputs)

From this, we can verify properties of the programs using inputs constrained in this way, using the property assured by this zkp.

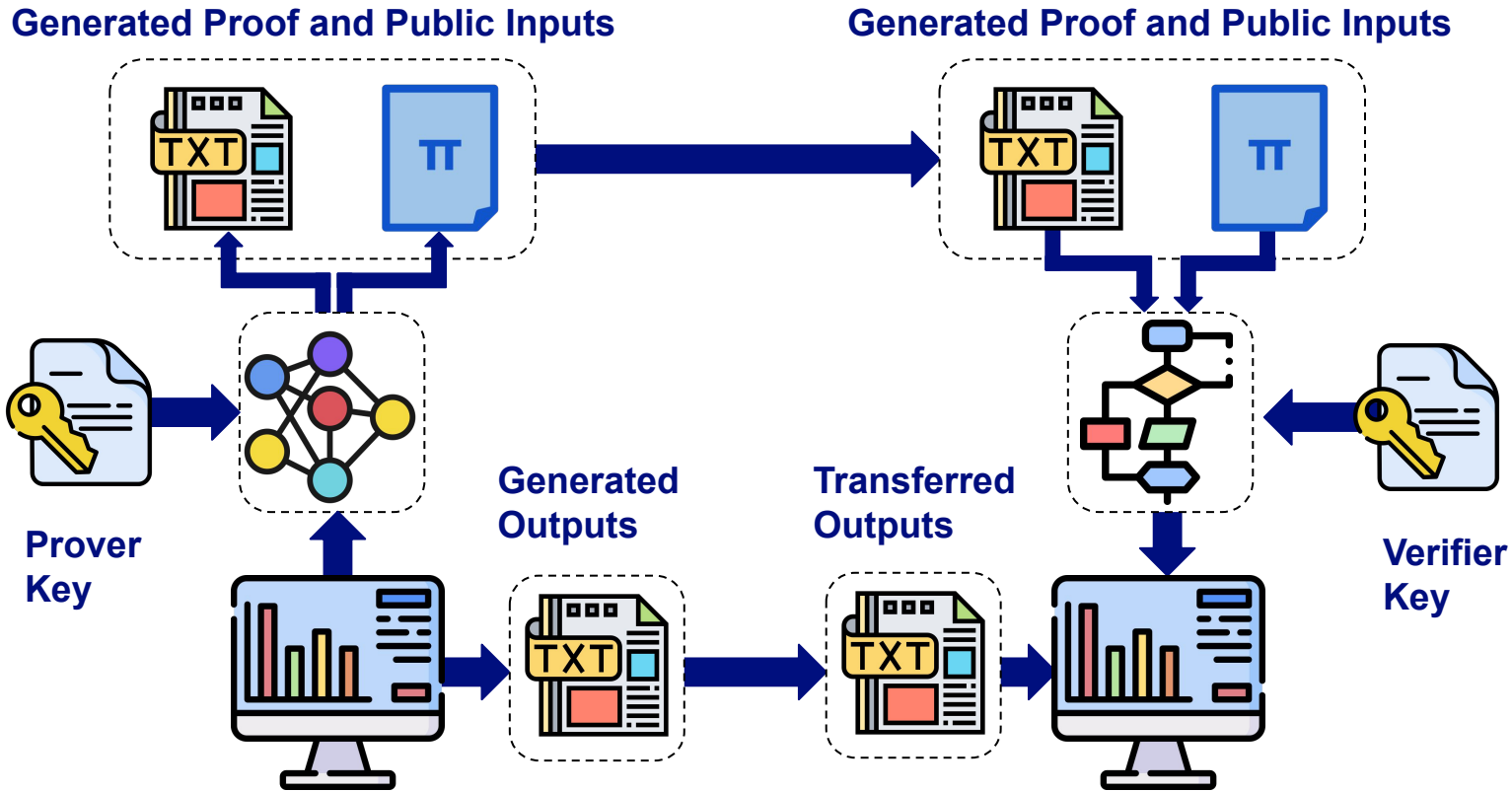
Solution: Assuring Safe Composition via zkSNARKs



Dependent Type Replacement by ZKPs

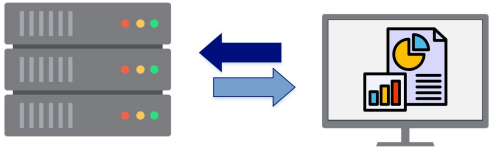


Distributed Verification

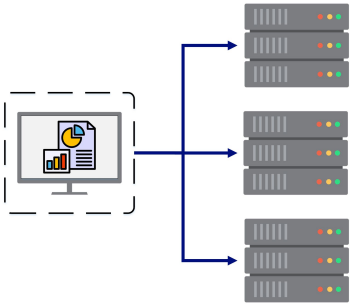


Benefits & Capabilities

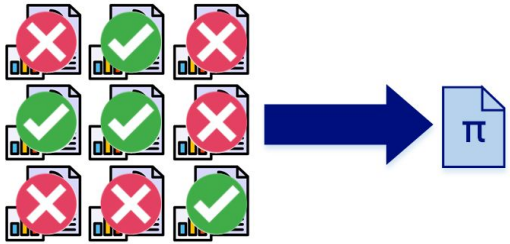
Using zero-knowledge proofs, we can combine cyber systems while preventing certain incorrect and malicious behaviors relating to mismatched outputs and input constraints.



ZKPs enforce system compatibility without the expense of manually proving correctness



Portable proofs artificially extends our trusted computing base beyond just our own system



ZKPs give fine-grained control over which bits of information to keep secret and which to reveal

Demo:
Verifying an RSA Encryption Pipeline

Background: Textbook RSA Cryptography

Encryption: $$\mathsf{Enc}_{k_{pub}}(msg) = msg^{k_{pub}} \bmod N = c$$

Decryption: $$\mathsf{Dec}_{k_{priv}}(c) = c^{k_{priv}} \bmod N = msg$$

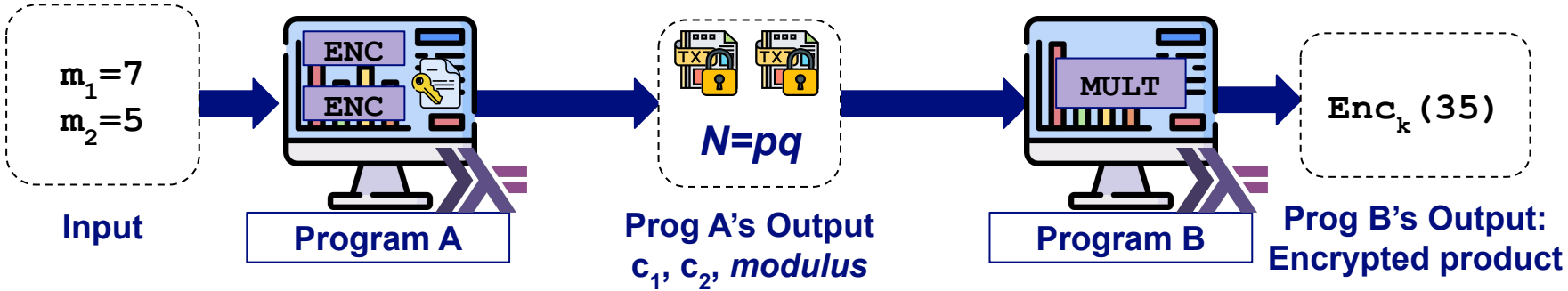
RSA is multiplicatively (×) homomorphic, meaning that if we encrypt two messages with the same key and modulus, the multiplication of those two ciphertexts equals the encryption of the multiplication of the plaintexts

$$\begin{aligned} c_1 * c_2 &\equiv msg_1^{k_{pub}} * msg_2^{k_{pub}} \bmod N \\ &\equiv (msg_1 * msg_2)^{k_{pub}} \bmod N \\ &\equiv \mathsf{Enc}_{k_{pub}}(msg_1 * msg_2) \end{aligned}$$

Demo: RSA Encryption Pipeline

Sample Pipeline

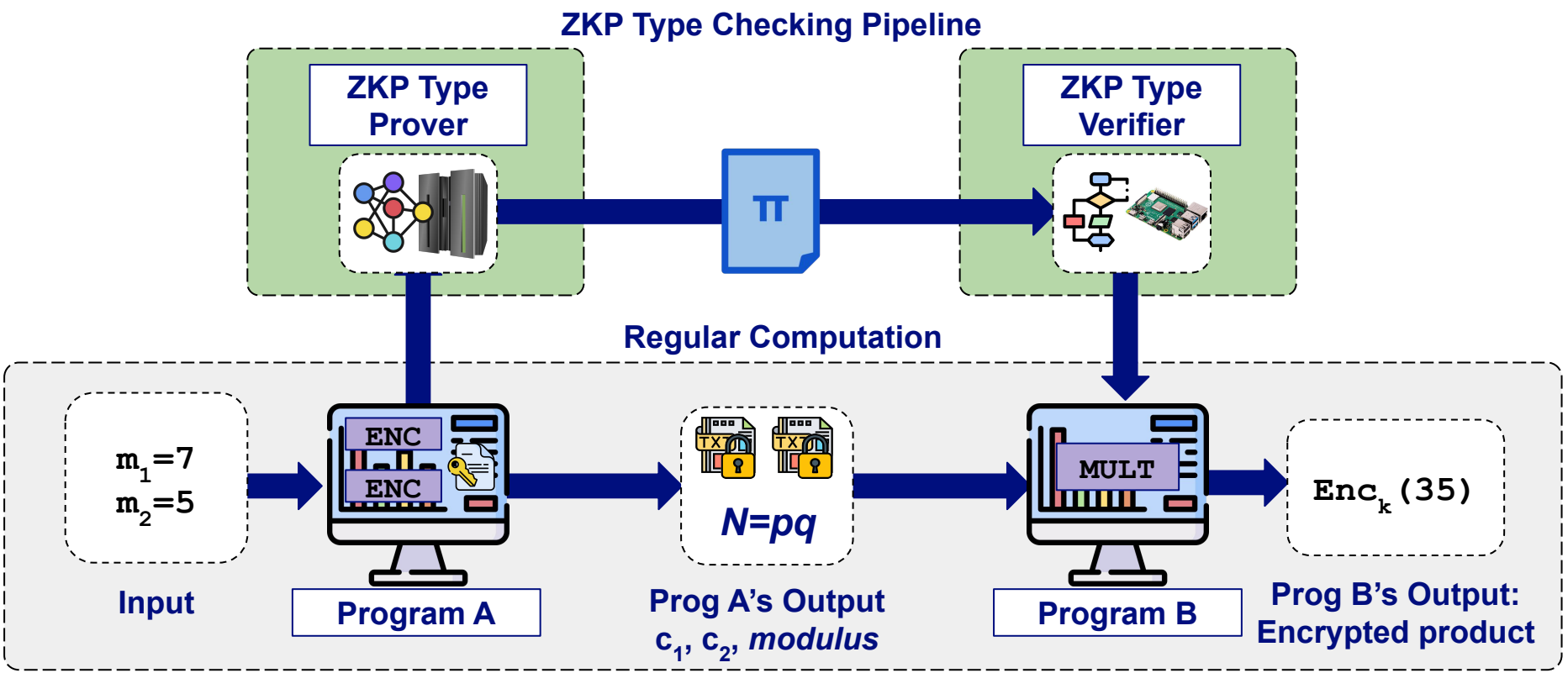
1. Program A encrypts two secret messages using RSA
2. Program B receives encrypted messages and multiplies them



Challenge

If we implement **A** and **B** in Haskell, program **B** can't guarantee it is multiplying valid RSA ciphertexts. **B** could end up yielding garbage and would be an **error** a type checker could catch **IF** it could see everything 1) only discoverable at runtime and 2) under the covers of encryption.

Demo: Encryption Pipeline with Type Checking ZKPs



Demo: Proving Type Checks with ZKPs

Haskell's type checker can't verify the encrypted variable's type until program runtime. Instead, we instruct it to know to ask for a ZKP of its type later.

Example. Type for a valid pair of RSA ciphertexts

```
ValidRSAPair : Set
ValidRSAPair =
  ∃[ ( bitLength, message1, message2, key, modulus,
       ciphertext1, ciphertext2) ] (
  RSAEncrypt key modulus message1 ≡ ciphertext1,
  RSAEncrypt key modulus message2 ≡ ciphertext2)
```

We can encode this proof as the `ValidRSAPair` type above, and generate a zkSNARK that proves type compliance using our compiler toolchain.

We can use Type-Level Haskell to generate redacted and un-redacted types, so type information is not lost between function calls, but sensitive information is not present.

Demo: Unredacted Pair Multiplier

```
multiplyPair ::
(Length, Message, Message, Key, Mod, CipherText, CipherText)
-> IO Message
multiplyPair r@(bits,m1,m2,pubKey,modulus,c1,c2) = do
  c1'  <- encrypt m1 key modulus
  c2'  <- encrypt m2 key modulus
  shouldBe (c1',c2') (c1,c2)
  let prod = (c1 * c2) `mod` modulus
  return prod
```

A non-redacted multiplication function input reveals sensitive information

Demo: Unredacted Pair Multiplier

```
multiplyPair ::
(Length, Message, Message, Key, Mod, CipherText, CipherText)
-> IO Message
multiplyPair r@(bits,[redacted],[redacted],[redacted],modulus,c1,c2) = do
  e1' <- encrypt [redacted] [redacted] modulus
  e2' <- encrypt [redacted] [redacted] modulus
  shouldBe (e1',e2') (e1,e2)
  let prod = (c1 * c2) `mod` modulus
  return prod
```

A non-redacted multiplication function input reveals sensitive information

Demo: Redacted Pair Multiplier

```
multiplyPair ::
Redacted RSAPair
-> IO Message
multiplyPair r@(bits,_,_,_,modulus,c1,c2) = do
  verifyZKP r
  prod <- (c1 * c2) `mod` modulus
  return prod
```

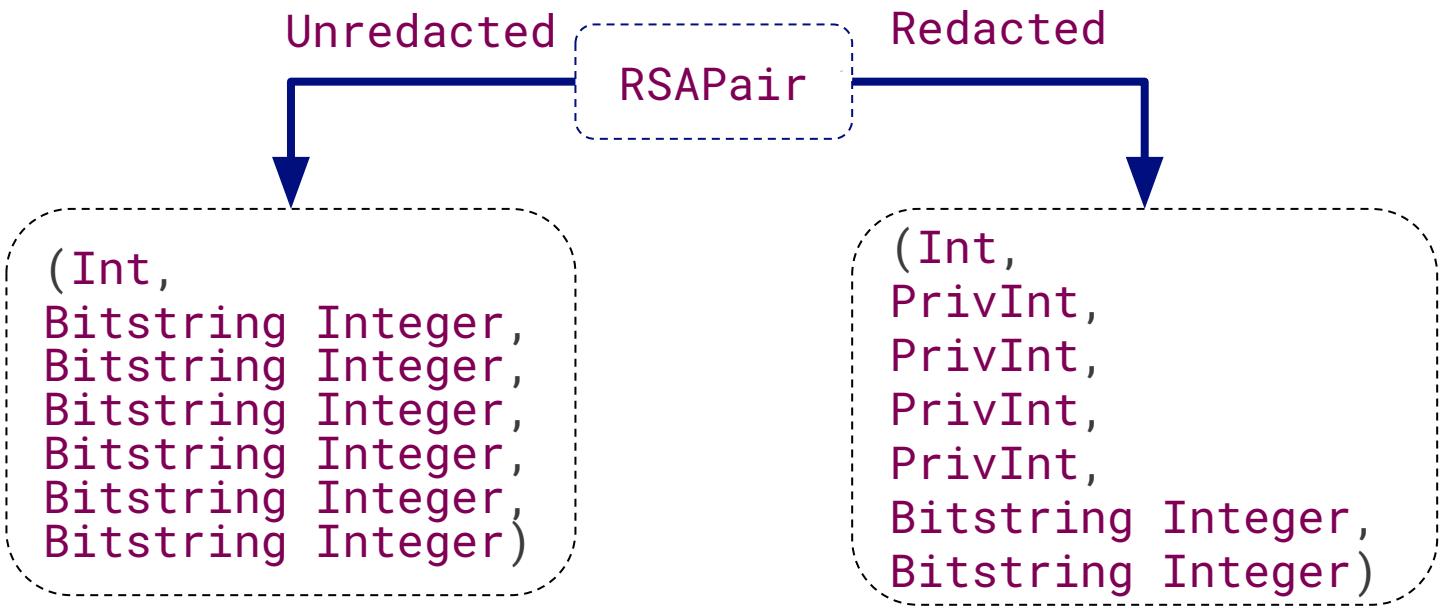
The redacted information is simply not available when passed as an input.

Encryption Property Verified with ZKP

```
encryptMessagePair ::
Length
-> Message
-> Message
-> PublicKey
-> Modulus
-> IO (Unredacted RSAPair)
encryptMessagePair bits m1 m2 key modulus =
   c1 <- encrypt m1 key modulus
   c2 <- encrypt m2 key modulus
   prepareZKP (bits, m1,m2, key, modulus, c1, c2)
```

This prepares the ZKP, which generates the proof files and redacts the information we don't want the other function to see.

Type-Level Programming Gives Type Safety



We can use type level programming to generate input and output types for functions from a central type.

A function to Illustrate Homomorphic Property

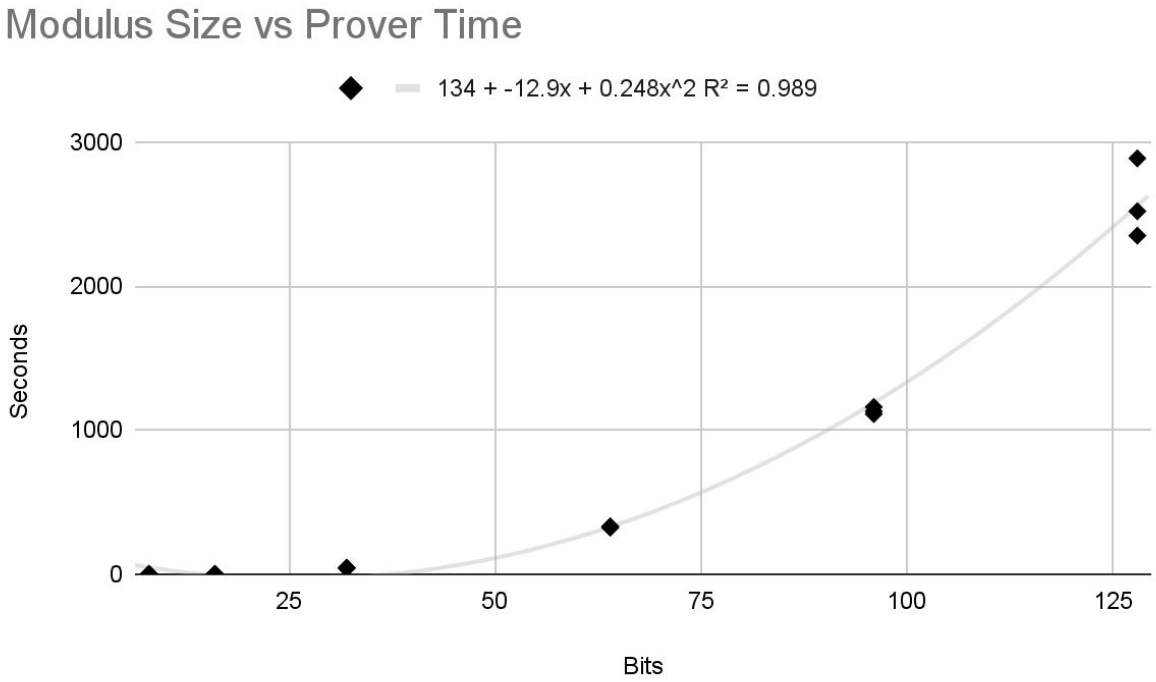
```
encryptMessagePair ::
Length
-> Message
-> Message
-> PublicKey
-> Modulus
-> IO (Unredacted RSAPair)
```

```
multiplyPair ::
Redacted RSAPair
-> IO Message
```

This function encrypts two messages with the same key and modulus, and returns them along with the bit width.

The decrypt function relies on the fact that the two supplied ciphertexts are encrypted with the same key and modulus.

Benchmarks for Demo



Demo Summary and Challenges

We were able to show an example of this approach using zkSNARKS to verify both functions interacting in a program, and programs interacting across a file system.

It relies on the writing of zkSNARK gadgets, which using extant libraries is extremely labor-intensive and requires knowledge of esoteric programming techniques.

In order to leverage the Haskell type system this approach requires type level Haskell programming, which is considered somewhat niche even among advanced Haskell programmers.

Can we mitigate these challenges?

Prototype ZKP Compiler

zkSNARK Construction for Program Verification [BCGTV13]

```
int myFunction(int a) {
     int b=a*a-4;
     return 3*b+a;
}
```

Rank-1 Constraint System (R1CS):

$$S \cdot A * S \cdot B = S \cdot C$$

1	0	1	0	1	0
a	1	a	1	a	1
t0	0	t0	0	t0	0
b	0	b	0	b	0

libsnark

- Computation
- Arithmetic Circuit
- R1CS

Proof Representation Of Program Execution

- QAP — Zero Knowledge Added
- LPCP — Succinctness Added
- LIP — Interactivity Removed
- zkSNARK

libsnark backend

Verifier Binary

Prover Binary

zkSNARK for Program Integrity

R1CS constraints

R1CS stands for Rank 1 Constraint system, a way of representing programs as sets of satisfiable constraints. This is represented by sets of equations of the following form, where ○ represents the dot product, S is a solution vector, A and B are vectors of variables, and each of the members of the vectors is a finite field element:

$$S \circ A \times S \circ B = S \circ C$$

In practice, mapping of variables and satisfying assignments is done automatically by zkp libraries, and thus are represented in the form:

$$\Sigma[\text{operand}] \times \Sigma[\text{operand}] = \Sigma[\text{operand}]$$

R1CS constraint example: Addition

To capture the addition of two field elements, x and y , which is assigned to the variable z, this can be represented in r1cs as

$$\Sigma[x,y] \times \Sigma[1] = \Sigma[z]$$

To make this hold we must supply a witness value for z,
in this case (x + y).

Note that this means we get multiple additions in a single constraint.

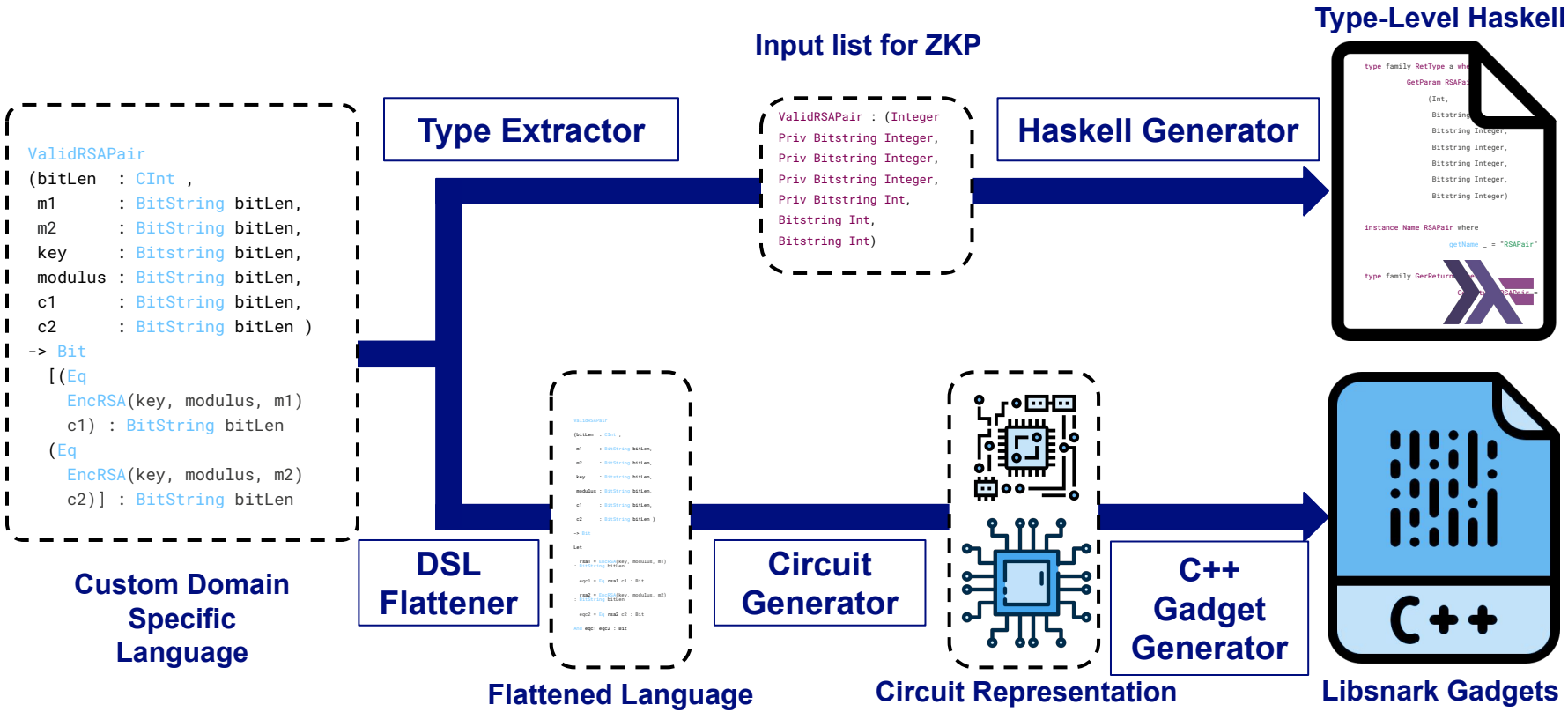
Limits of R1CS

The basic unit of memory in R1CS is a finite field element. Computation can be done directly on them, however these are limited in size, and many operations (such as comparison or modulo) can't be performed directly on them.

We can use these R1CS constraints to build circuits that use binary representations of numbers, one bit per field element, to do bitstring-level computations.

As these gadgets effectively tie together R1CS constraints, inputs and outputs are explicit, leading naturally to a to functional programming paradigm.

Full Compiler Pipeline



Demo: Custom Domain Specific Language Example

```
ValidRSAPair
(bitLen  : CInt ,
 m1      : Priv BitString bitLen,
 m2      : Priv BitString bitLen,
 key     : Priv Bitstring bitLen,
 modulus : BitString bitLen,
 c1      : BitString bitLen,
 c2      : BitString bitLen )
-> [Bit,Bit]
[
  (bsEq [EncRSA(key, modulus, m1), c1], : BitString bitLen)
  (bsEq [EncRSA(key, modulus, m2), c2]] : BitString bitLen)
]
```

Demo: Flattened Language Example

```
ValidRSAPair
(bitLen  : CInt ,
 m1      : Priv BitString bitLen,
 m2      : Priv BitString bitLen,
 key     : Priv Bitstring bitLen,
 modulus : BitString bitLen,
 c1      : BitString bitLen,
 c2      : BitString bitLen )
-> [Bit,Bit]
Let
  [rsa1] = EncRSA[bitLen,key,modulus,m1] : BitString bitLen
  [eqc1] = bsEq[rsa1,c1] : Bit
  [rsa2] = EncRSA[bitLen,key,modulus,m2] : BitString bitLen
  [eqc2] = bsEq[bitlen,rsa2,c2] : Bit
[eqc1,eqc2] : Bit
```

Demo: Using polymorphic functions

```
bsEq
(bitLen  : CInt ,
 bs1     : BitString bitLen,
 bs2     : BitString bitLen)
-> [Bit]
Let
  [eqList] = zipWith[eq, bs1, bs2] : BitString bitLen
  [acc]    = I
  [eqBit]  = foldl[bitLen,and,acc,eqList]   : Bit
[eqBit] : Bit
```

Type checker

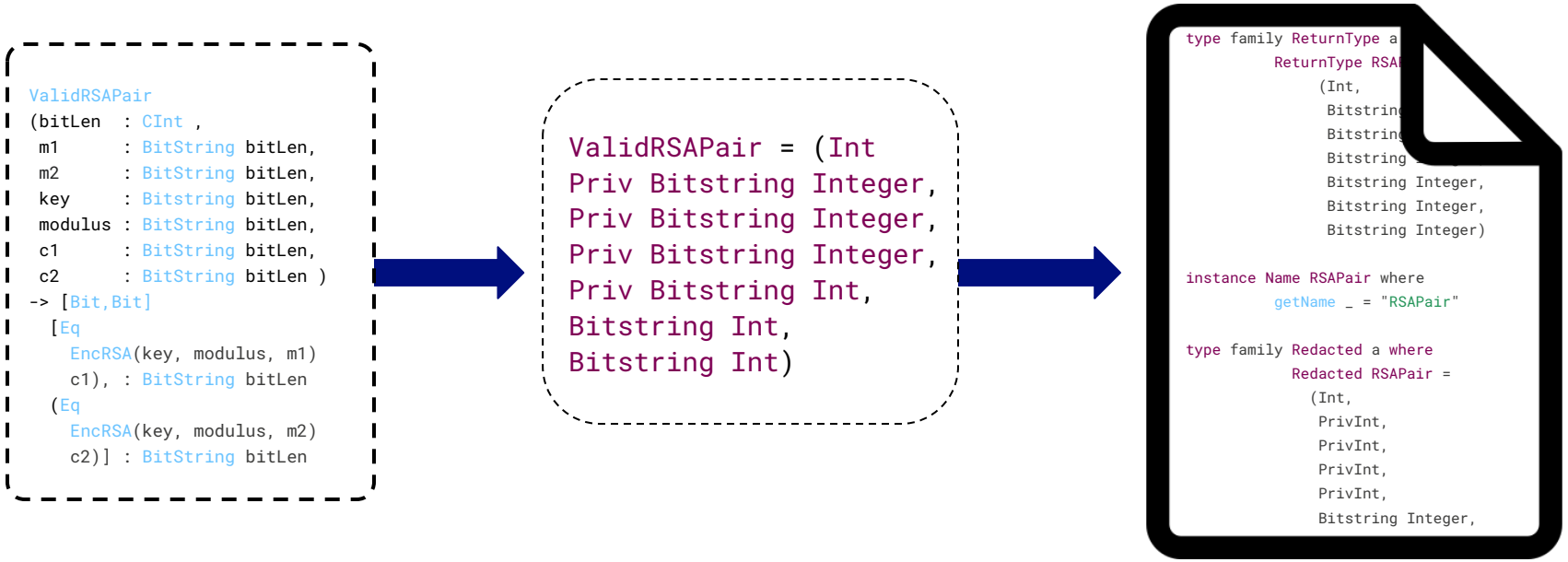
```
Typechecking error in
fun ICOS(x0 : Bit) -> [Num]
  Let
    [_intcons0] = -128
    [x1] = CADD(x0,_intcons0) : [Num]
    [y0] = ISIN(x1) : [Num]
  in
    [y0]
:while trying to unify input args:
[C NumT,C NumT] and [C BitT,C NumT]
NumT and BitT do not unify
```

One of the benefits of using a DSL is understandable error messages.

We implemented a type checker that supports polymorphism, and supplying functions as arguments.

Using this type checker, we caught bugs in our prototype gadgets.

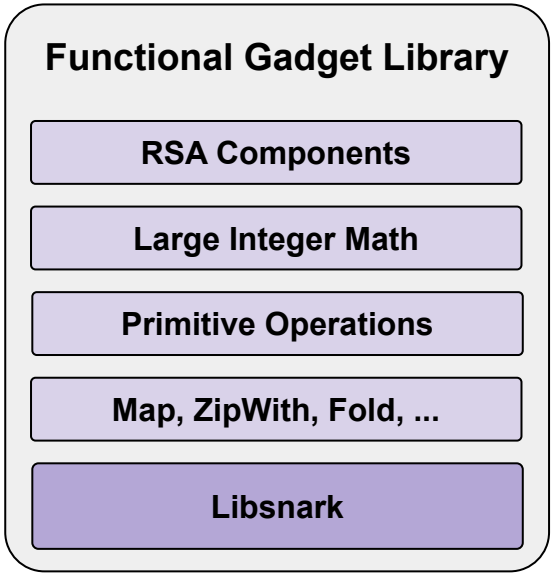
Type-level Haskell Generation Step



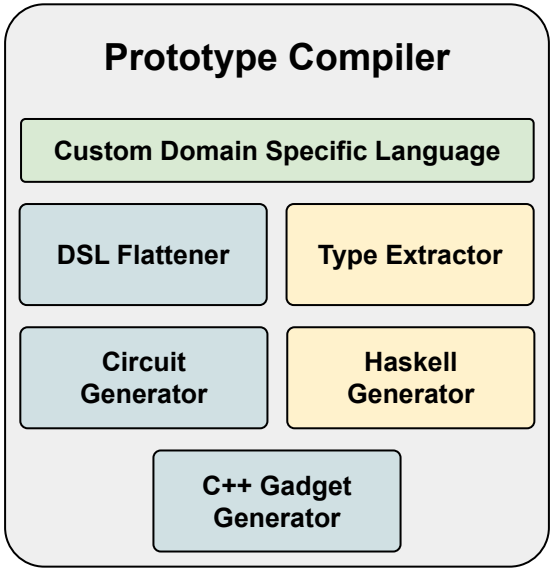
Expression **zkSNARK Type** **Type-Level Haskell**

Implementation Summary

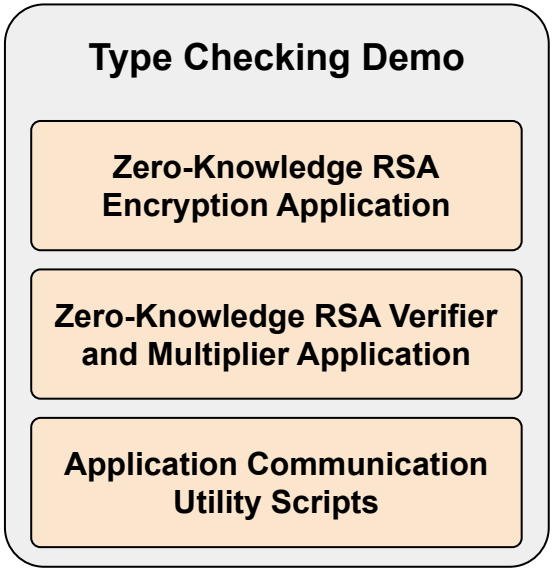
We developed a *library* of zkSNARK *gadgets* and *types* in `C++` using `Libsnark`



We developed a *custom compiler* in `Haskell` to apply *functional programming techniques* to zkSNARK development



We produced a demo *dependently-typed* zkSNARK application for RSA *encryption* and *verification*



Conclusion

By using zkSNARKs to prove that values have specific **dependent types**, it is possible to provably assure compatibility and correctness without revealing sensitive information and extend our trusted computing base well beyond our own system.

The approach we developed expands the scope of what non-interactive zero-knowledge proofs can capture to include properties about both the execution and correctness of programs

Future Work

I. Increase the extent of Haskell language integration to **enforce verification on a programming language level** rather than trusting programmers to run the verifier binaries manually.

II. Leverage approach to work with several ongoing efforts at LANL to help **verify mission-relevant cyber systems** that utilize sensitive information

III. Build more advanced compiler **automation** to automatically integrate type-level haskell and compile libsnark programs to allow faster development times.

IV. Build **optimization** steps to reduce number of gates into the compiler, and optimize existing gadgets.

V. Increase the expressivity of the language to include ZKPs for **uncertainty measures** and **machine learning model properties** developed by fellow LANL student, Zachary DeStefano (A-4).

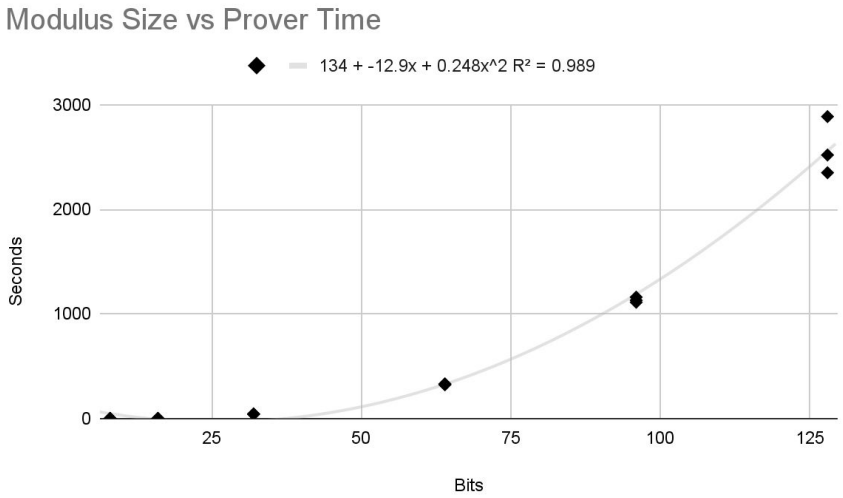
VI. Move the DSL into a monad which admits computation failure, to distinguish between gadgets that always succeed at witness generation and those which can fail.

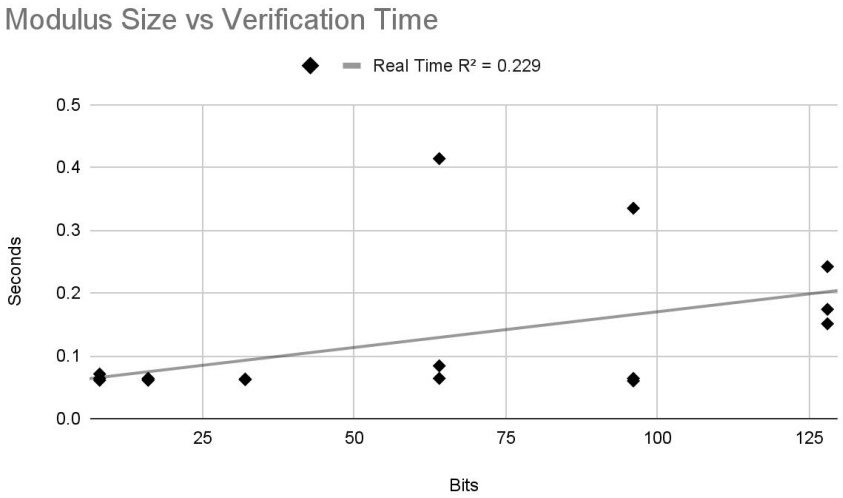
Questions?

dbarrack@lanl.gov

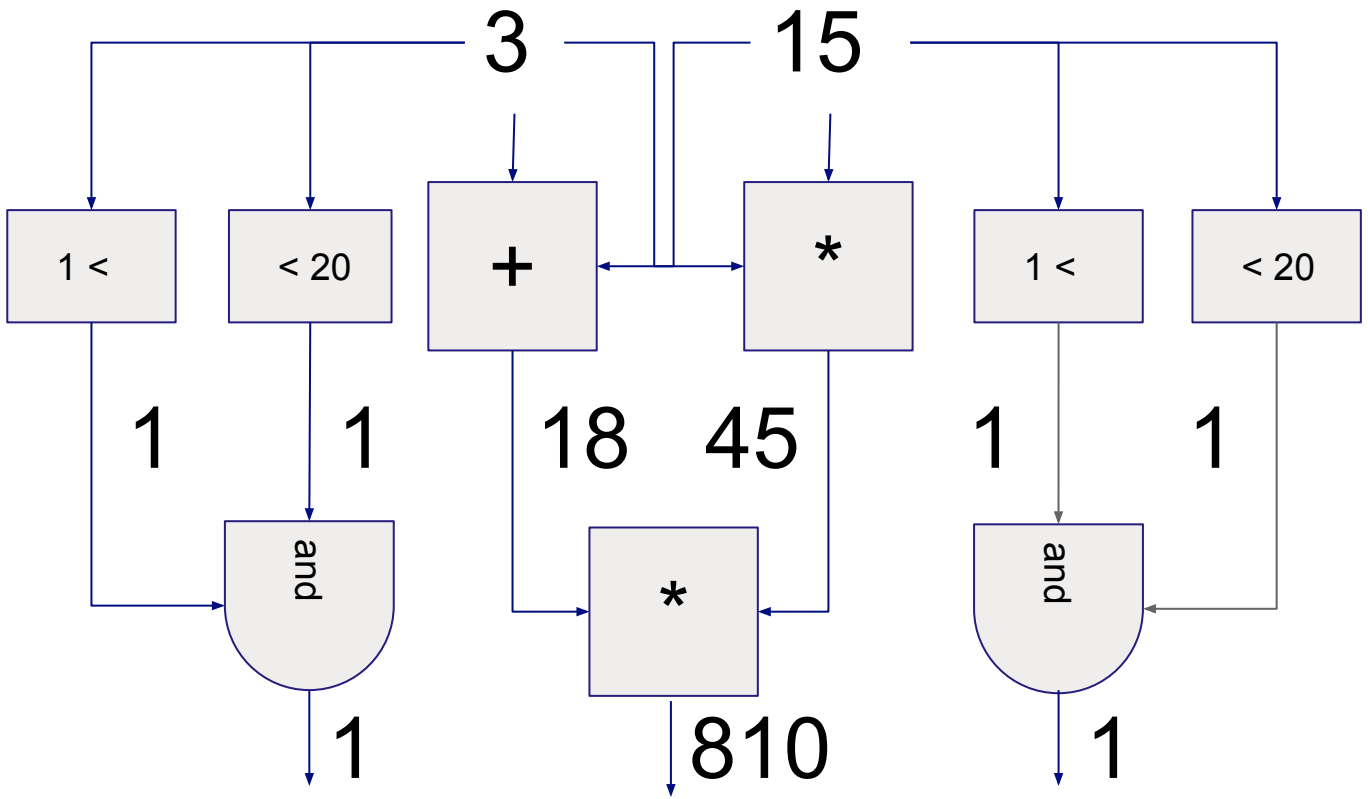
Backup

Benchmarks for Demo





Our function as a circuit



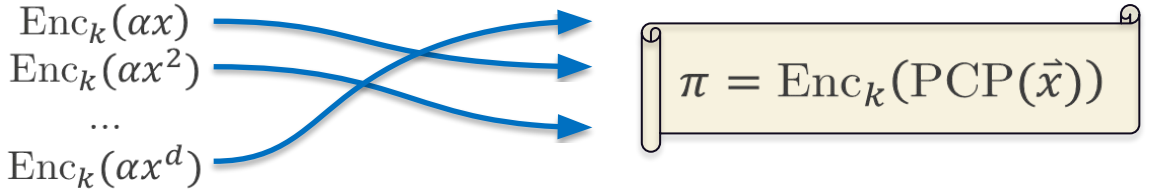
Theory Behind ZKPs (Backup)

Circuit Evaluation

1. Publish homomorphically encrypted building blocks for a program

$$\mathrm{CRS} = \{\mathrm{Enc}_k(\alpha x), \mathrm{Enc}_k(\alpha x^2), \ldots, \mathrm{Enc}_k(\alpha x^d)\}$$

2. Prover blindly re-assembles them to compute the desired circuit (e.g. an evaluation of the PCP circuit) and adding random blinds where appropriate



3. Verifier checks content by simply decrypting

$$\mathrm{Dec}_k\big(\mathrm{Enc}_k(\mathrm{PCP}(\vec{x}))\big) = \begin{cases} 1 & \text{if valid} \\ 0 & \text{if invalid} \end{cases}$$

Language Grammar

$$\langle \textit{Bit Literal} \rangle = \mathrm{I} \mid \mathrm{O}$$

$$\langle \textit{Arg} \rangle = \mathrm{Op}\ \langle \textit{String} \rangle \mid \mathrm{Var}\ \langle \textit{String} \rangle$$

$$\langle \textit{Literal} \rangle = \mathrm{Num}_v\ \mathbb{N} \mid \mathrm{Bit}_v\ \langle \textit{Bit Literal} \rangle$$

$$\begin{aligned} \langle \textit{Expr} \rangle ::= {} & \mathrm{Call}\ [\langle \textit{Templates} \rangle]\ [\langle \textit{Arg} \rangle]\ [\langle \textit{SimpleTy} \rangle] \\ \mid {} & \langle \textit{Literal} \rangle \end{aligned}$$

$$\langle \textit{TopEx} \rangle ::= \mathrm{Let}\ [([\mathrm{id} : \langle \textit{SimplyType} \rangle], \langle \textit{Expr} \rangle)]\ [\mathrm{id}]$$

$$\langle \textit{Fun} \rangle ::= \mathrm{Fun}\ \langle \textit{String} \rangle\ [\mathrm{id}]\ [(\mathrm{id}, \langle \textit{Ty} \rangle)]\ [\langle \textit{SimpleType} \rangle]\ \langle \textit{TopEx} \rangle$$

$$\langle \textit{Program} \rangle ::= \mathrm{Program}\ \langle \textit{String} \rangle\ [\langle \textit{Fun} \rangle]$$

Language grammar

$$
\begin{aligned}
\langle \mathit{SimpleType} \rangle ::= \ & \text{Bit} \\
| \ & \text{Num} \\
| \ & \text{List } \langle \mathit{SimpleType} \rangle \ \langle \mathit{Length} \rangle \\
| \ & \text{Bool} \\
| \ & \text{FixPt} \\
| \ & \text{Int} \\
| \ & \text{TypeVariable}
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathit{Type} \rangle ::= \ & \langle \mathit{SimpleType} \rangle \\
| \ & [\langle \mathit{SimpleType} \rangle] \rightarrow [\langle \mathit{SimpleType} \rangle]
\end{aligned}
$$

$$
\langle \mathit{Function\ Type} \rangle = [\text{Template}]\ [\langle \mathit{Type} \rangle]\ [\langle \mathit{SimpleType} \rangle]
$$

Big step semantics

$$\frac{\Gamma \vdash fun :< t'_1 ... t'_n > [ty'_1 .. ty'_m] \to [ty''_1 ... ty''_o] \quad \Gamma \vdash (a_1, ... a_m) : [ty_{a1}, ..., ty_{am}] \quad U(< t_1, ..., t_o >, < t'_1 ... t'_n >) = S \quad S[ty''_1 ... ty''_o] = [ty_1, ..., ty_o] \quad S[ty'_1 .. ty'_m] = [ty_{a1}, ..., ty_{am}]}{\Gamma \vdash fun < t_1, ..., t_n > (a_1, ..., a_n) : [ty_1, ..., ty_o]}$$

$$\frac{\Gamma \vdash e : [t_1, .., t_n] \quad \Gamma \vdash var_1 : t_1, ... var_n : t_n \vdash \text{Let } vs\ e\ :\ t_{rs}}{\Gamma \vdash \text{Let } ([var_1, ..., var_n] = e_l\ ::\ vs)\ rs\ :\ t_{rs}}$$

$$\frac{\Gamma \vdash vs : [t_1, .., t_n]}{\Gamma \vdash \text{Let } []\ vs\ :\ [t_1, ..., t_n]}$$

$$\frac{\Gamma \vdash p_1 : t_1, ..., p_n\ :\ t_n,\ topEx\ :\ [t_{o1}, ..., t_{on}}{\Gamma \vdash \text{def } name\ < t_1, ..., t_2, > (p_1 : t_1, ..., p_n : t_n)\ \text{TopEx}\ :\ [t_{o1}, ..., t_{on}]}$$

Big step semantics

$$\frac{}{Bit : \text{Type}_s} \quad \frac{}{Int : \text{Type}_s} \quad \frac{}{Num : \text{Type}_s} \quad \frac{\Gamma \vdash c : t \qquad t : \text{Type}_s \qquad \Gamma \vdash len : \text{Int}}{\text{List } c\ len : \text{Type}_s}$$

$$\frac{\begin{array}{ll}\Gamma \vdash [a_0, ..., a_n] : [t_{a0}, ..., t_{an}] & \Gamma \vdash [t_{a0}, ..., t_{an}] : [\text{Type}_s, ..., \text{Type}_s] \\ \Gamma \vdash [b_0, ..., b_m] : [t_{a0}, ..., t_{an}] & \Gamma \vdash [t_{b0}, ..., t_{bm}] : [\text{Type}_s, ..., \text{Type}_s]\end{array}}{[a_0, .., a_n] \to [b_0, ..., b_n] : \text{Type}_c}$$

$$\frac{}{\Gamma \vdash\ I : Bit} \quad \frac{}{\Gamma \vdash\ O : Bit} \quad \frac{}{\Gamma \vdash\ Num_v\ i : Num} \quad \frac{\Gamma \vdash x : t \in \Gamma}{\Gamma \vdash\ Var\ x : t}$$

$$\frac{x : as \to bs \in \Gamma}{\text{OP } x : as \to bs} \quad \frac{\Gamma \vdash x : t}{\Gamma \vdash lit_v\ x : [t]}$$