LA-UR-21-26873
Approved for public release; distribution is unlimited.

Title: Secure System Composition and Type Checking using Cryptographic Proofs

Author(s): Barrack, Daniel Abraham

Intended for: For discussion with other ZKSnark researchers

Issued: 2021-07-16

Los Alamos NATIONAL LABORATORY

Secure System Composition and Type Checking using Cryptographic Proofs

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

Mentor: Michael J. Dixon
Co-Mentor: Boris Gelfand

June 21st, 2021

ISTI Information Science & Technology Institute

Portland State UNIVERSITY

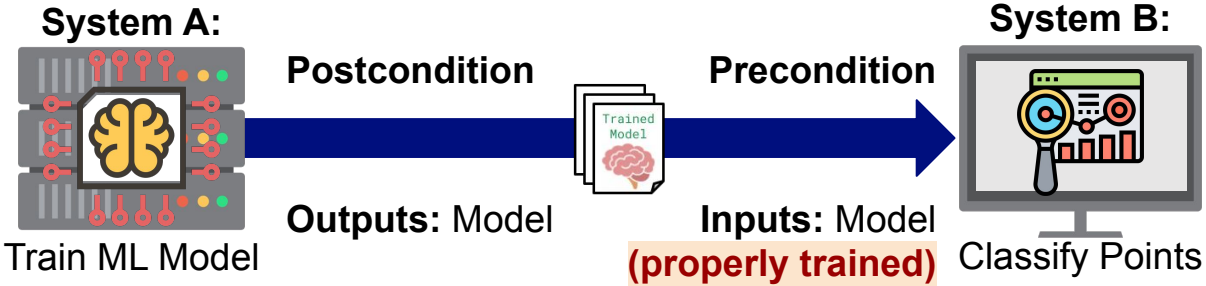
Challenge: Formally Verifying System Composition

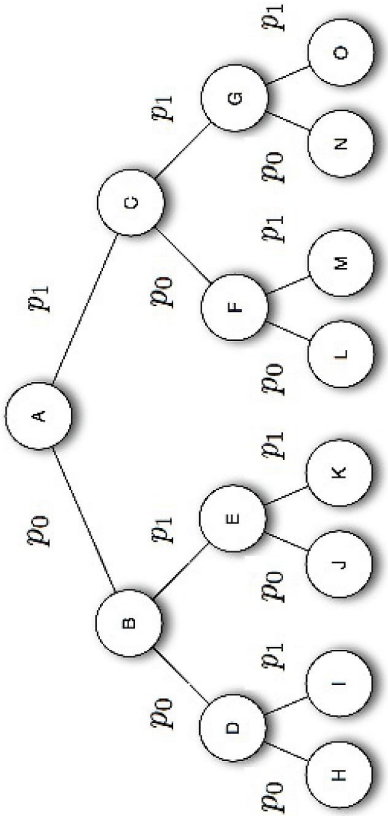
We can use formal methods to verify that systems compose correctly without the possibility of incorrect behavior.

This means exhaustively checking that System A's postconditions agree with System B's preconditions. If so, it is safe to compose.

Normal Setting: Every computational path must be accounted for and checked. Verification cost (time) is **multiplicative** across systems.

$$\text{Cost} = |S_1| * |S_2| * \ldots * |S_n|$$





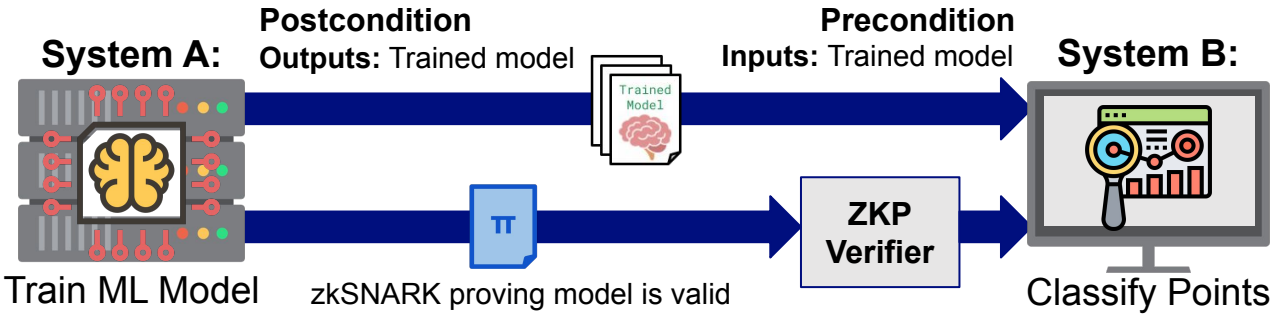
Solution: Assuring Safe Composition via zkSNARKs

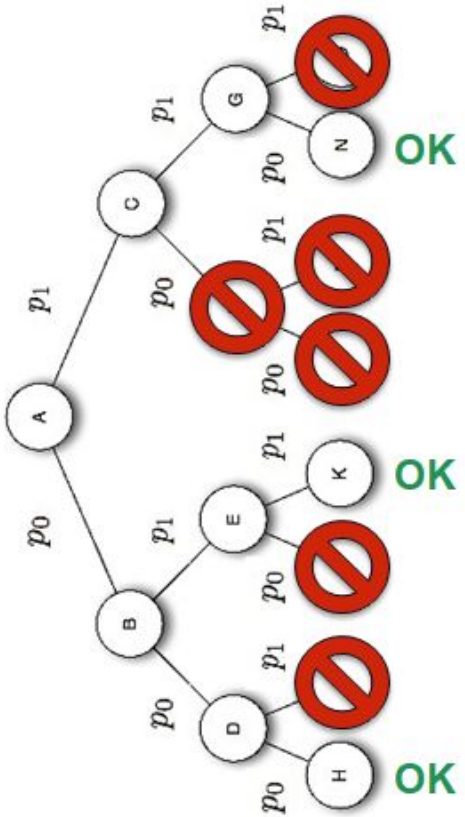
Zero-knowledge proofs can be used to provide type checking guarantees of input/output properties without exposing secrets.

Verification can be done modularly so that the cost is **additive**.

$$\text{Cost} = |S_1| + |S_2| + \ldots + |S_n|$$

Bad proofs and inputs can still exist, but now are cryptographically (exponentially) hard to find and exploit.





Preconditions and Postconditions with Types

```
Type MLModel =
     (w : Weights, error(w) < 0.05, log : AuditLog, execute(log) == w)
trainModel : (x : [Input]) -> MLModel
```

```
classifyPoint : (y : Input, model : MLModel) -> 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.

Preconditions and Postconditions with Types

```
Type MLModel =
     (w : Weights, error(w) < 0.05, zkp : ZKP, log : AuditLog, execute(log) == w)
trainModel : (x : [Input]) -> MLModel
```

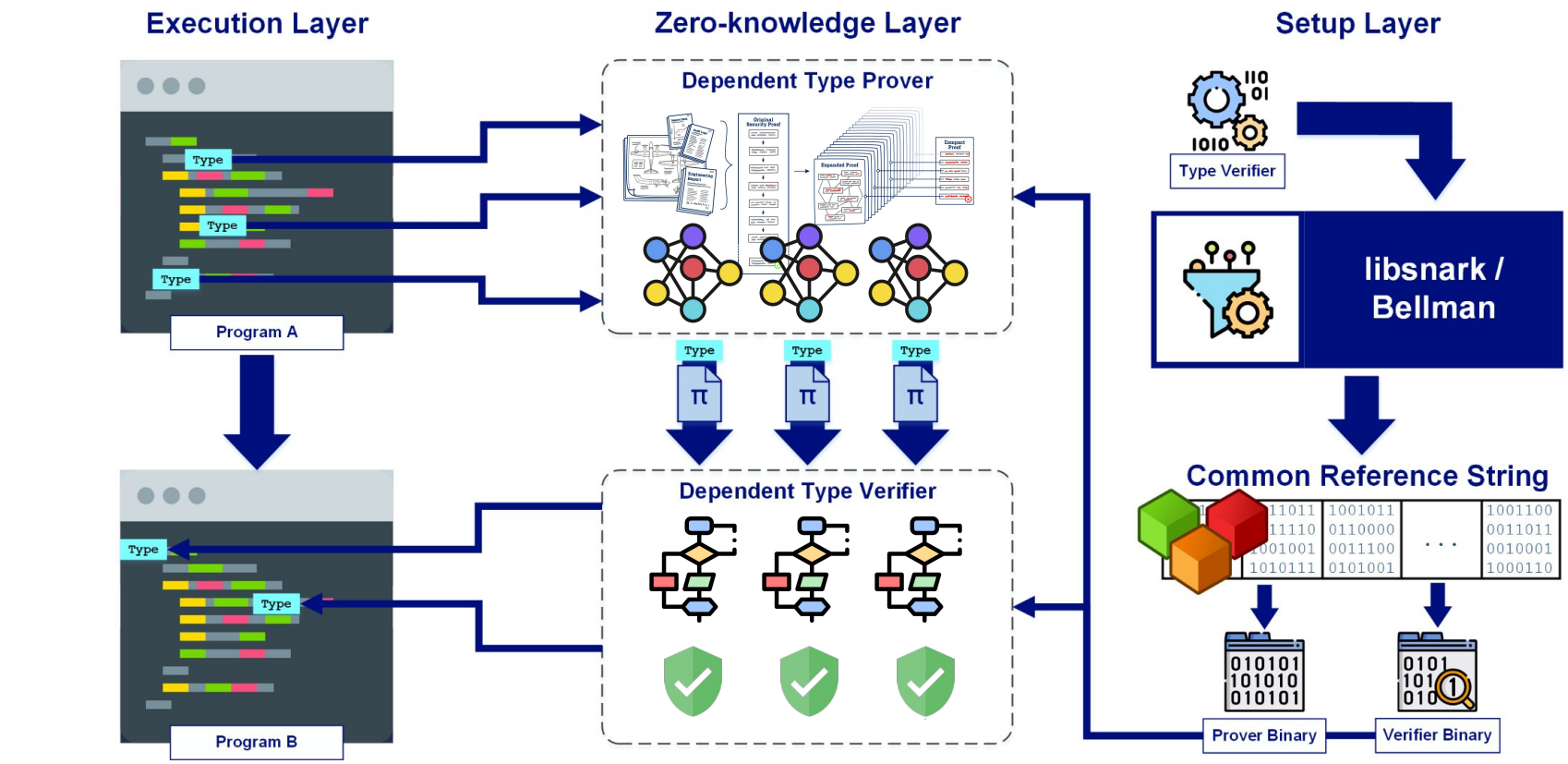
```
classifyPoint : (y : Input, model : MLModel, verif : ZKPVerifier) -> IO Class
```

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.

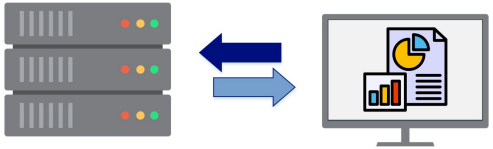
The audit log and its proof would be **huge**. Instead, we can use a super small zkSNARK to prove this dependent type and pass it along instead. We only need to handle the case where the check fails.

Dependent Type Replacement by ZKPs

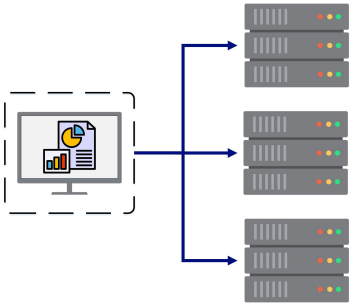


Benefit & Capability

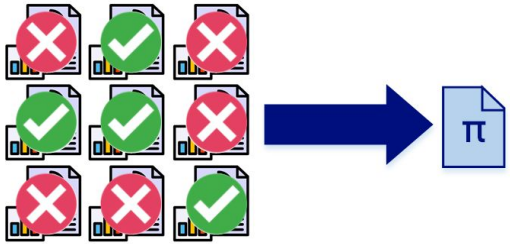
Using zero-knowledge proofs we can combine systems while preventing certain incorrect behaviors relating to mismatched outputs and input constraints.



ZKPs enforce system compatibility without the expense of manually proving correctness



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



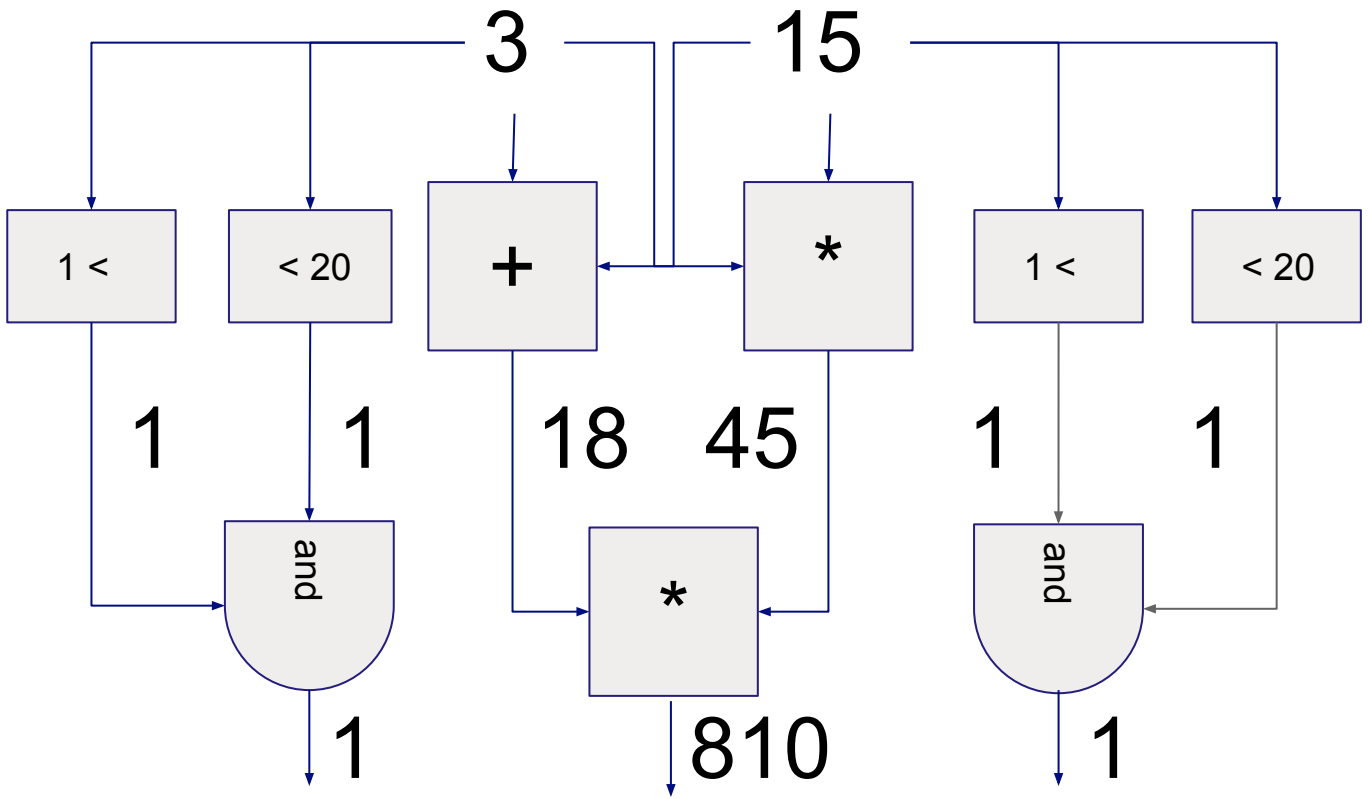
ZKPs give fine-grained control over which bits of information to keep secret and which to prove

Project Roadmap

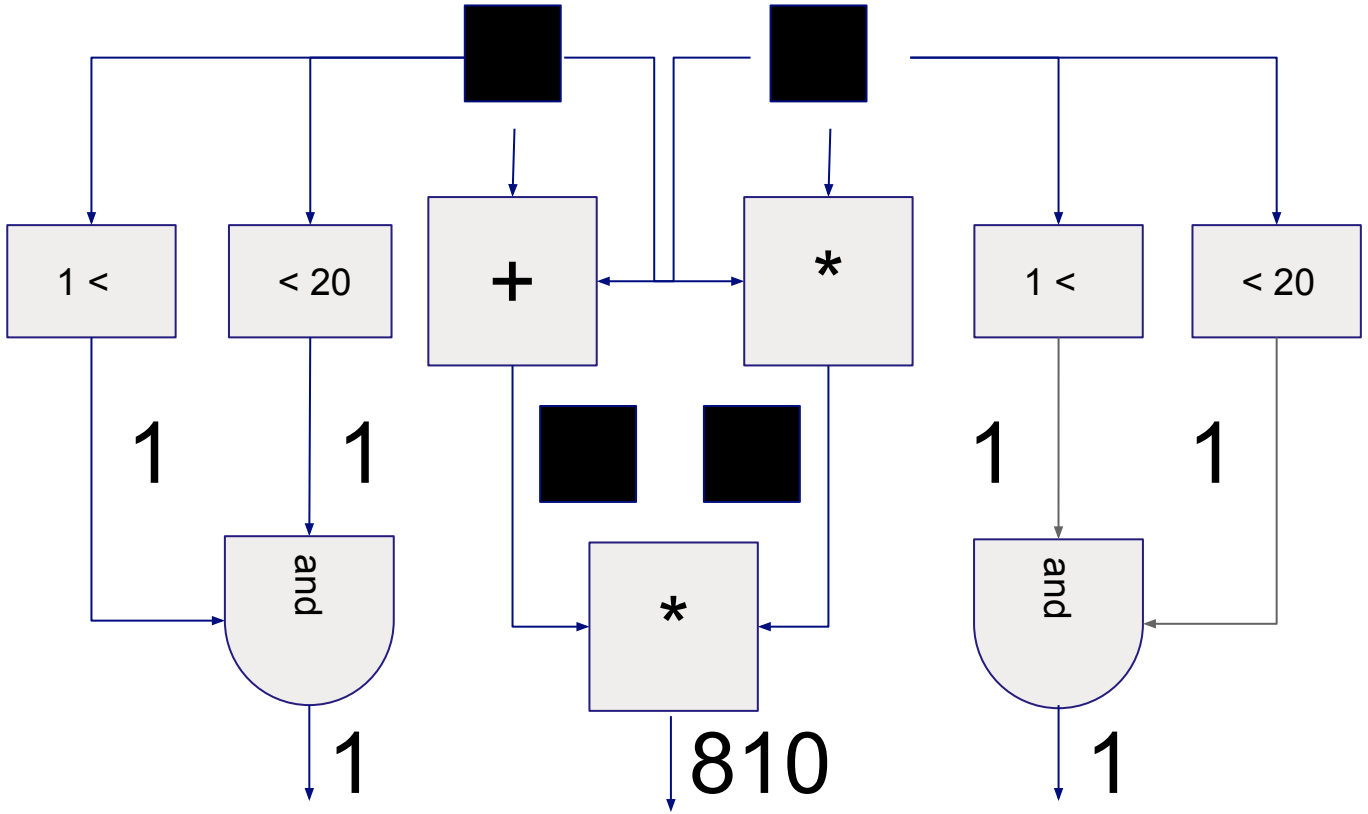
1. Prototype and test program interoperability
   - Manually implement skeleton code in place of zero-knowledge proofs to interact with example program
2. Implement constraint related ZKP gadgets
   - Constraints capture type information that is immediately useful to the test program
3. Develop and set up prototype demonstrations
   - Replace skeleton code with handcrafted ZKP gadgets
   - Develop prototype compiler to read type annotations from file and generate constraints
4. Benchmark and Evaluate

Backup

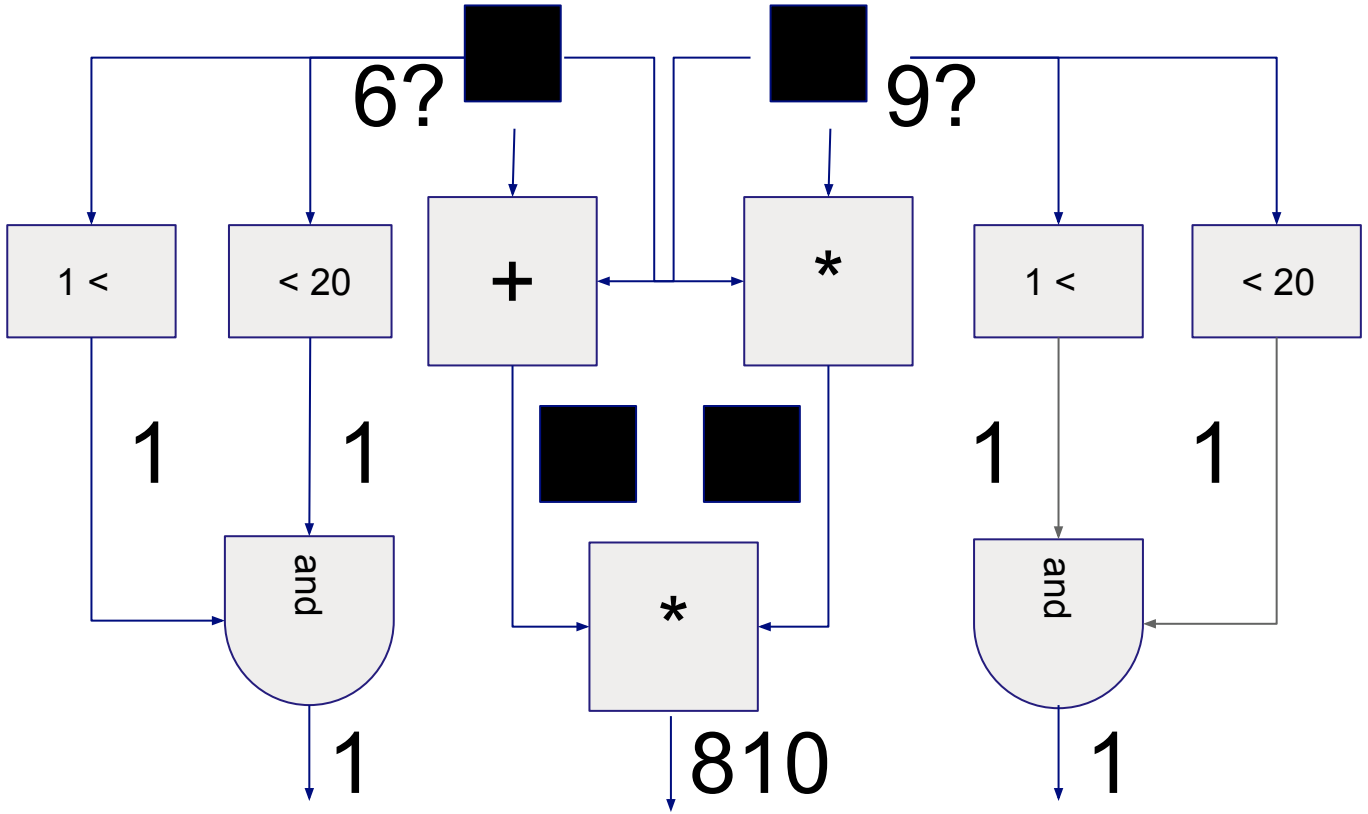
Our function as a circuit



A redacted circuit with zkSNARKs



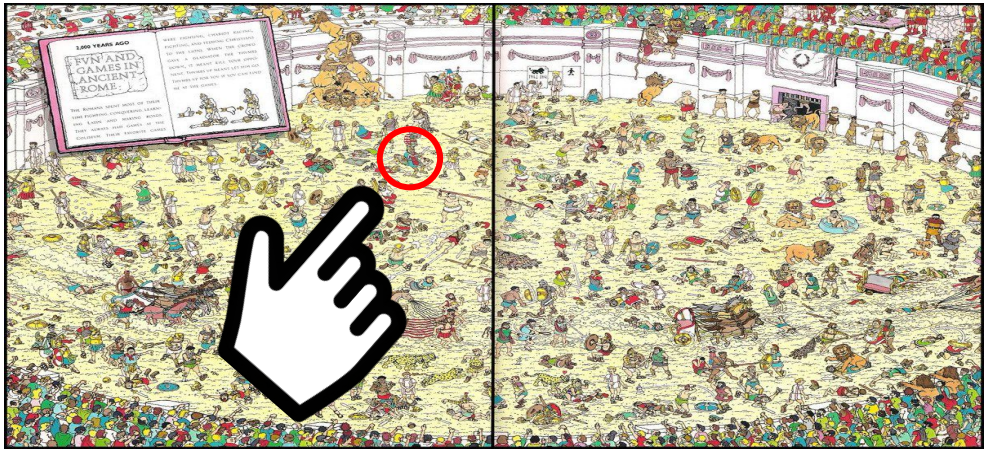
A redacted circuit with zkSNARKs



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



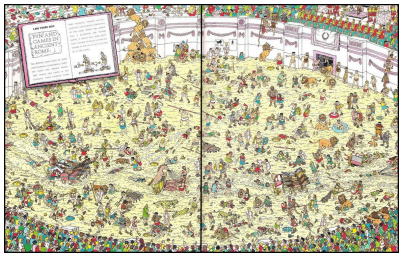
- Not zero-knowledge!

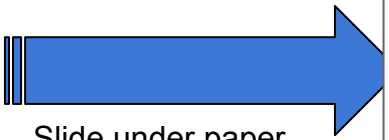
  This kind of proof leaks all information about his location, much more than simply that you have *knowledge* of the location

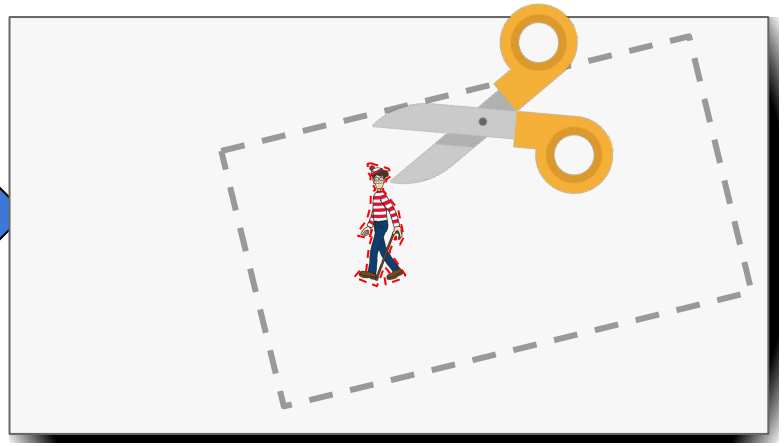
Zero-Knowledge Proof for *Where's Waldo?*

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







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

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

Completeness vs. Soundness

Typical proof systems have 100% completeness and 100% soundness

Completeness: ℙ[true statement AND verifier accepts] = 1
"Everything true is provable"

Soundness: ℙ[false statement AND verifier rejects] = 1
"False statements aren't provable"

Cryptographic Proof Systems

Cryptographic proof systems have variable completeness and soundness. For non-interactive zero-knowledge proofs we care about:

(Completeness) ℙ[true statement AND verifier accepts] = 1
"Everything true is provable"

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

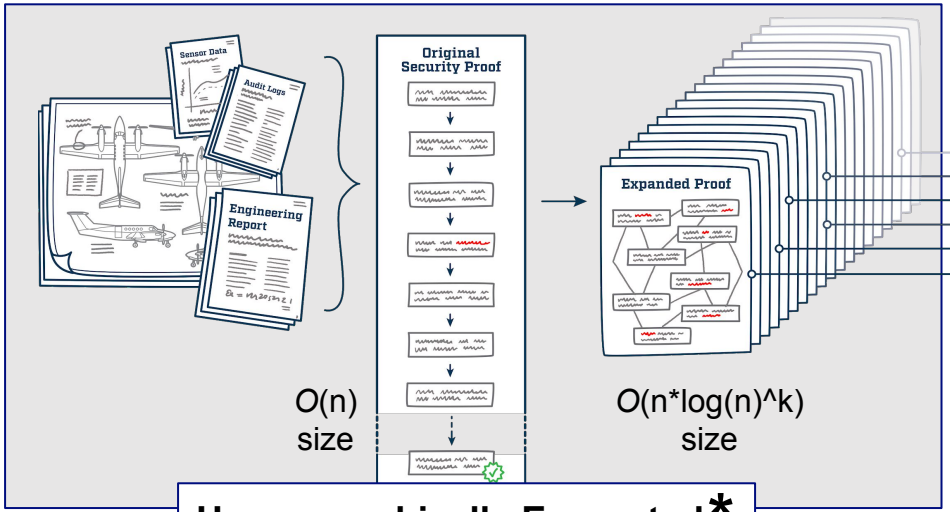
We sacrifice minimal amount of soundness (have to break crypto to produce counter-example) in order to get valuable proof properties

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

Theory Behind ZKPs (Backup)

PCPs & Hardness of Approximation

Intuition

Efficient approximation scheme for a problem implies that it an easy to create a good enough looking “fake” solution (witness)

So,

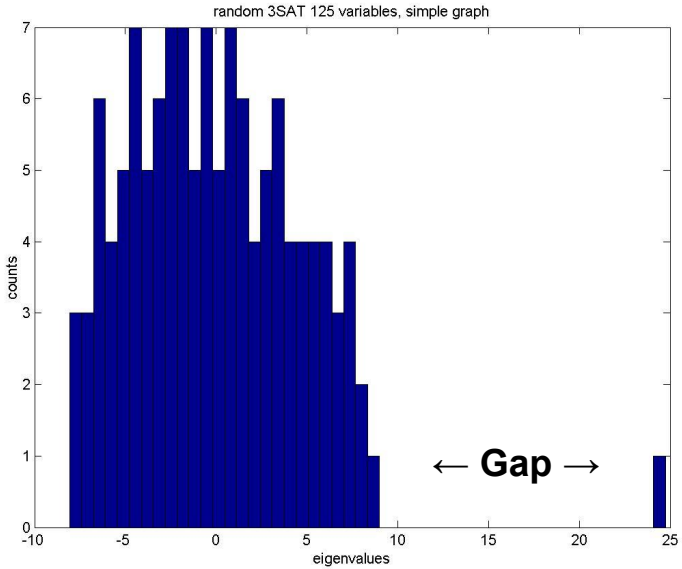
Hard to approximate

⇔

Hard to create a convincing fake witness that appears optimal

⇔

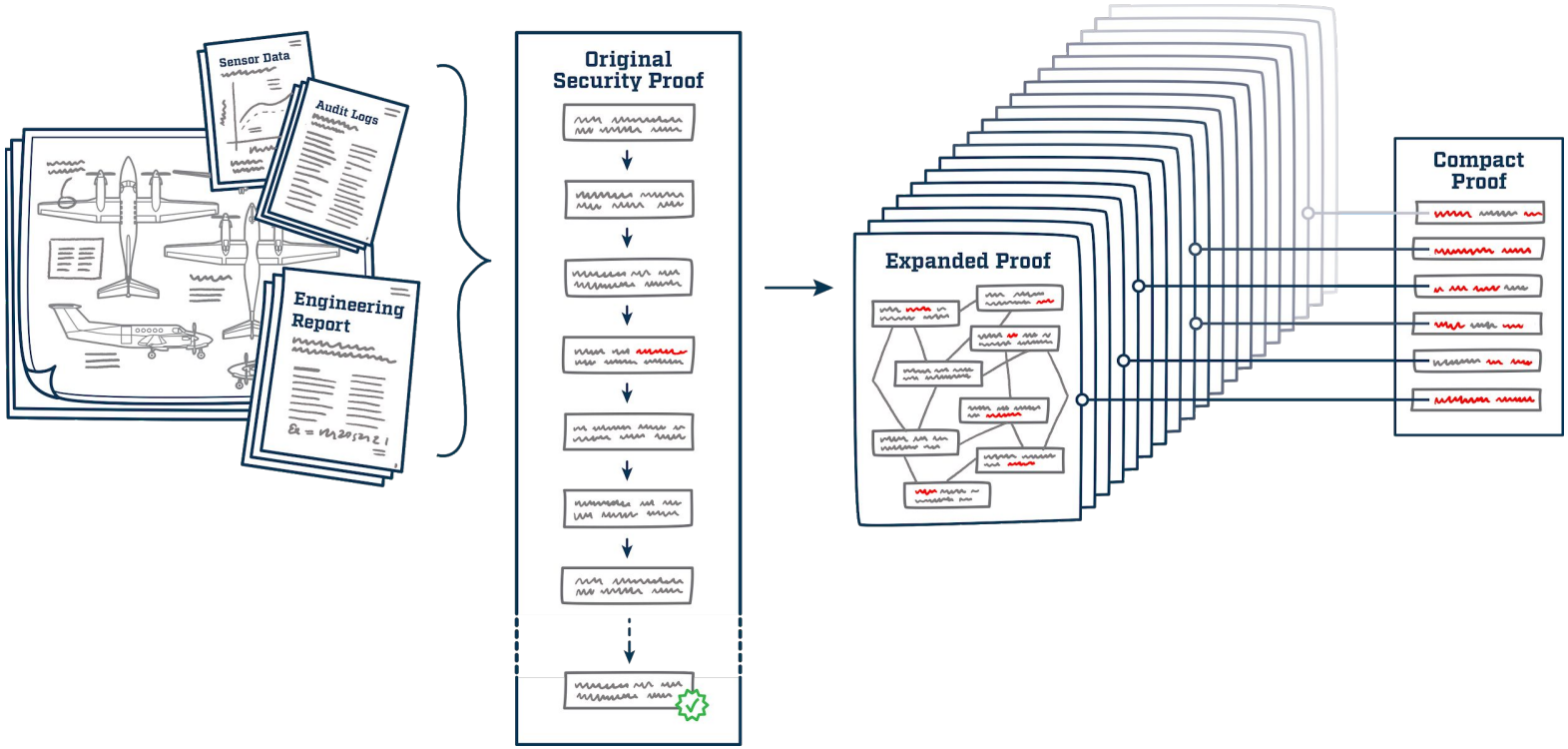
Good witnesses imply that best solutions exist



Easy to find

Hard to find

NIZK Overview

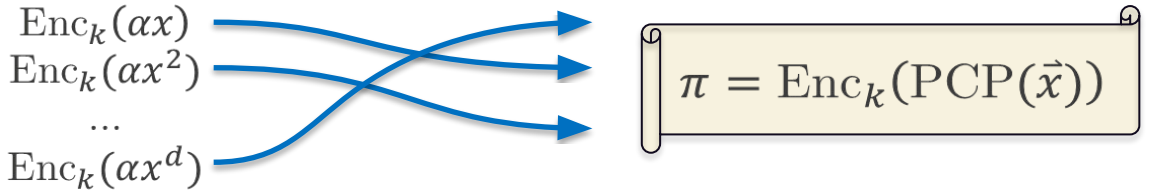


Circuit Evaluation

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

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