

# ZK-SecreC: A Domain-specific Language for Zero-Knowledge Proofs

Raul-Martin Rebane

# Why a DSL?

- Bring ZK to new problem domains
- Simplify preparation of large statements
- Allow complex proofs
- Interface with many proof systems

# The information flow types

## Domain

A value may be

- known only to  $P$ , or
- known to  $P$  and  $V$ , or
- known at compile-time

## Stage

A value may be

- local to parties, or
- also present at circuit

# The information flow types

## Domain

A value may be

- known only to  $P$ , or @prover
- known to  $P$  and  $V$ , or @verifier
- known at compile-time @public

## Stage

A value may be

- local to parties, or \$pre
- also present at circuit \$post

$x$  : **int** \$pre @prover

# The information flow types

## Domain

A value may be

- known only to  $P$ , or `@prover`
- known to  $P$  and  $V$ , or `@verifier`
- known at compile-time `@public`

## Allowed classification

`@public`  $\rightarrow$  `@verifier`  $\rightarrow$  `@prover`

`$post`  $\rightarrow$  `$pre`

No declassification on domains!

## Stage

A value may be

- local to parties, or `$pre`
- also present at circuit `$post`

`x : int $pre @prover`

# The information flow types

## Domain

A value may be

- known only to  $P$ , or `@prover`
- known to  $P$  and  $V$ , or `@verifier`
- known at compile-time `@public`

## Allowed classification

`@public`  $\rightarrow$  `@verifier`  $\rightarrow$  `@prover`  
`$post`  $\rightarrow$  `$pre`

No declassification on domains!

## Stage

A value may be

- local to parties, or `$pre`
- also present at circuit `$post`

`x : int $pre @prover`

## `$pre` $\rightarrow$ `$post`

- Creates an input for the circuit
- There is an operation for this

`wire { x } : int $post @prover`

## Other similar languages..

- conflate `@prover` and `@verifier`
  - Loses information, which checks are necessary in the circuit
  - Note that `$pre @prover` and `$pre @verifier` have different integrity properties
- or**
- conflate `@verifier` and `@public`
  - Cannot separate the inputs defining the circuit from the verifier's inputs to the circuit

# ZK Programming is hard

- Writing proofs is non-trivial
  - Underconstrained circuits are dangerous
  - Optimizing is required
- ⇒ Don't need to shy away from ZK mechanics



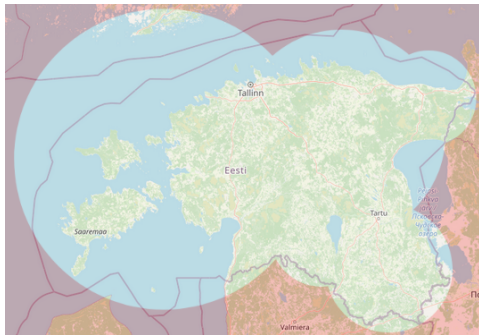
# Polymorphism

```
fn sum[N : Nat, $S, @D](xs : list[uint[N] $S @D]) -> uint[N] $S @D {  
    let mut s = 0;  
    for i in 0..length(xs) { s = s + xs[i] }  
    s  
}
```

# Interleaving

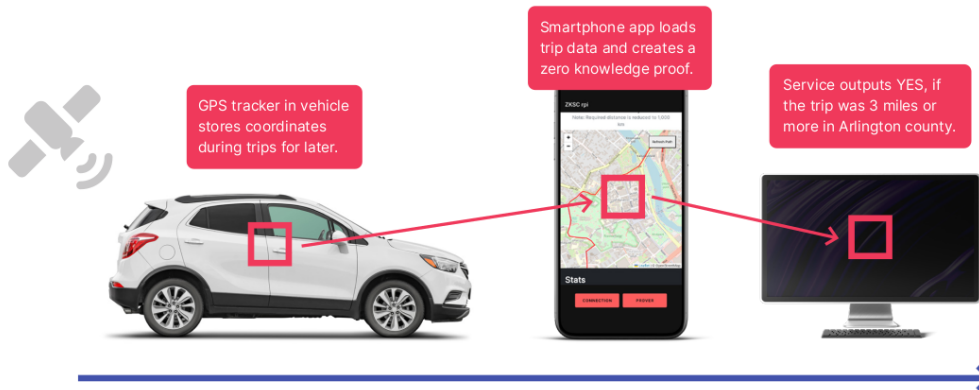
- Can interleave on- and off-circuit computation
- Valuable for complex statements
- FFI for Rust
- Requires an effect system

# Vehicle Subsidy Demo



- Grant terms: 80,000km travelled, 80% of it in Estonia
- Prove compliance without revealing GPS data
- Increased transparency to the user

# Vehicle Subsidy Demo



# Vehicle Subsidy Demo

Prover	Tracking duration	Runtime
Pixel 5a	Month	127s
Pi 5	Month	23s
Pi 5	Year	152s

Assuming 1h a day driving, 1 coordinate per 30sec.

# Changing protocols

We built using:

- Mersenne61 prime
- Arya inequalities
  - Verifier's challenges

Ligero supported:

- FFT-friendly prime
- No challenges

# Conditional compiling

Type predicates and branching for features

- Field switching
- Verifier's challenges
- Function gates
- Vectorization

# Backends

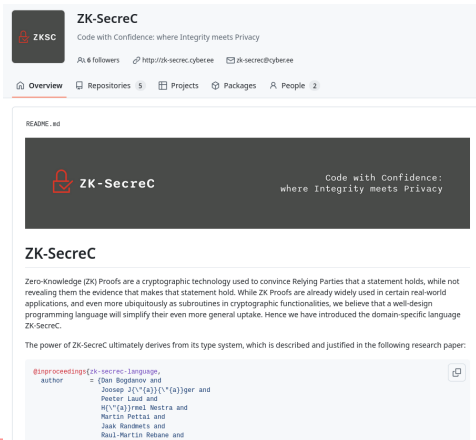
- Circom, SieveIR
- Galois Mac'n'Cheese
  - Field switching
  - Function gates
  - Vectorized calls
  - Verifier challenges
- emp-zk
- zkb++



# Language Integrations

- Standard library includes
  - Bigints
  - EC
  - JSON parsing
  - Fractional numbers
  - Poseidon
- Maximize code reuse in both domains
- Off-circuit: Rust FFI
- On-circuit: CirCom FFI (coming)

# Other Examples




**ZK-Secrec**  
Code with Confidence: where Integrity meets Privacy

14 followers · <http://zk-secrec.cybernetica.com> · [zk-secrec@cybernetica.com](mailto:zk-secrec@cybernetica.com)

Overview · Repositories 5 · Projects · Packages · People 2

README .md

 **ZK-Secrec** Code with Confidence:  
where Integrity meets Privacy

## ZK-Secrec

Zero-Knowledge (ZK) Proofs are a cryptographic technology used to convince Relying Parties that a statement holds, while not revealing them the evidence that makes that statement hold. While ZK Proofs are already widely used in certain real-world applications, and even more ubiquitously as subroutines in cryptographic functionalities, we believe that a well-design programming language will simplify their even more general uptake. Hence we have introduced the domain-specific language ZK-Secrec.

The power of ZK-Secrec ultimately derives from its type system, which is described and justified in the following research paper:

```
@inproceedings{zk-secrec-language,  
  author      = {Dan Bogdanov and  
                Josep J. G. (a)ger and  
                Peeter Laud and  
                H. (a)rnel Neutra and  
                Martin Pettai and  
                Jaak Randmeets and  
                Raul-Martin Rebane and  
                ...}
```

[github.com/zk-secrec/examples](https://github.com/zk-secrec/examples)

- Face recognition
- Server log audit
- Bank record audit
- Medical check

# Thank you

## Acknowledgments

- This research has been funded from DARPA's "Securing Information for Encrypted Verification and Evaluation (SIEVE)" program



[cybernetica](#)



[Cybernetica](#)



[cybernetica\\_ee](#)



[Cybernetica](#)