# Enforcing ideal-world leakage bounds in real-world secret sharing MPC frameworks

José Bacelar Almeida[1,4] Manuel Barbosa[1,3] Gilles Barthe[2] **Hugo Pacheco**[1,4] Vitor Pereira[1,3] Bernardo Portela[1,3]

July 10, 2018 @ CSF 2018

[1]INESC TEC, Portugal

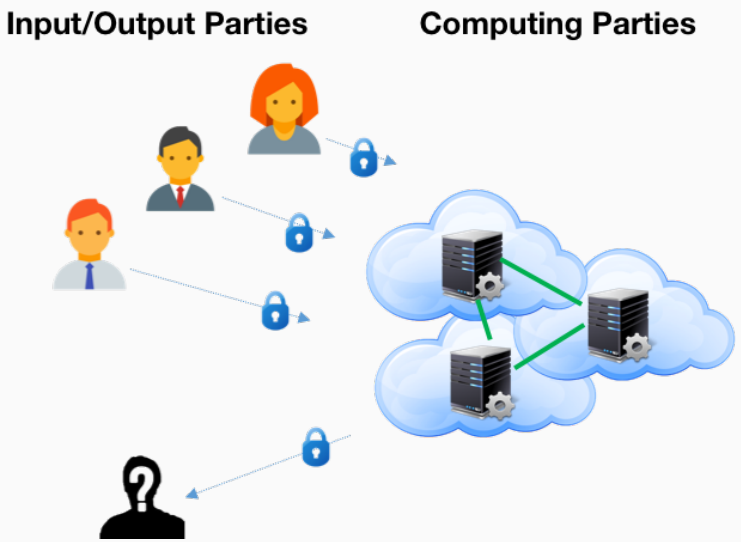
[2]IMDEA Software Institute, Spain

[3]FC Universidade do Porto, Portugal

[4]DI Universidade do Minho, Portugal

# Multi-Party Computation (MPC)

- powerful cryptographic paradigm
- allow two or more mutually distrusting parties to collaboratively compute over their private data, only revealing the result of the computation
- theoretical foundations laid almost 30 years ago
- recently growing for privacy-critical applications

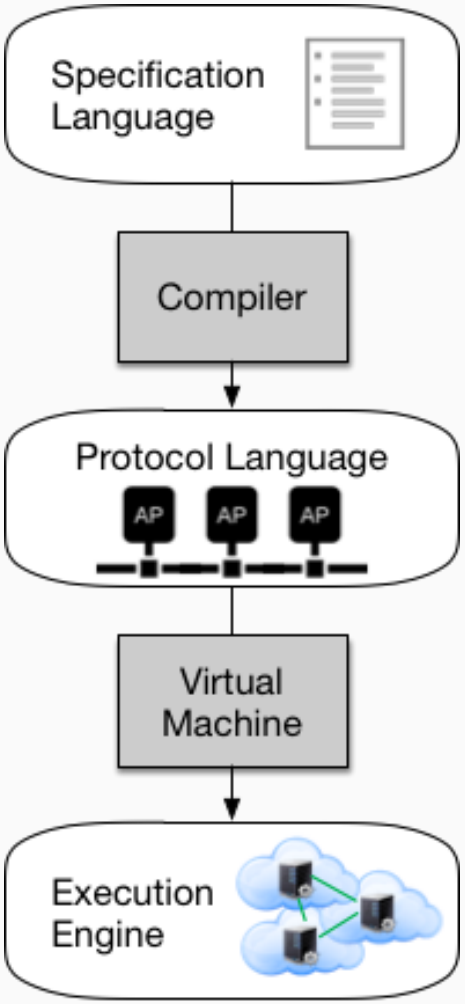


# MPC Software Stacks

- a few MPC frameworks in recent years (Sharemind, FRESCO and others)
- non-expert programmers develop MPC applications in "traditional" languages

```
uint maximum (uint[1] xs)
{
    uint m = xs[0];
    for (uint i=1; i<size(xs); i+=1)
        if (xs[i]>m) m = xs[i];
    return m;
}
```

- program compiled to sequence of secure MPC protocols for very simple tasks
- evaluation done by distributed virtual machine



# MPC Dilemma: eficiency *vs*...

- from previous slide: program $\xrightarrow{\text{compilation}}$ secure MPC protocols
- secure protocols for simple tasks:
  - add/mul, and/or, ...
- simple tasks are composable!
- however...
  - impractical to run the whole program obliviously
  - private control flow requires exploring all program paths
  - what about leaking control flow?

# MPC Dilemma: ...*vs* security

- practical languages
  - information flow type system
  - MPC-specific public control-flow restrictions
  - special `declassify` operation
- good performance still requires a MPC expert
  - which values to declassify? at which security cost?
- how much does this program leak?

```
secret maximum (secret xs) {
  secret m = xs[0];
  for (public i=1; i<size(xs); i+=1)
  if declassify(xs[i]>m) m = xs[i];
  return m; }
```

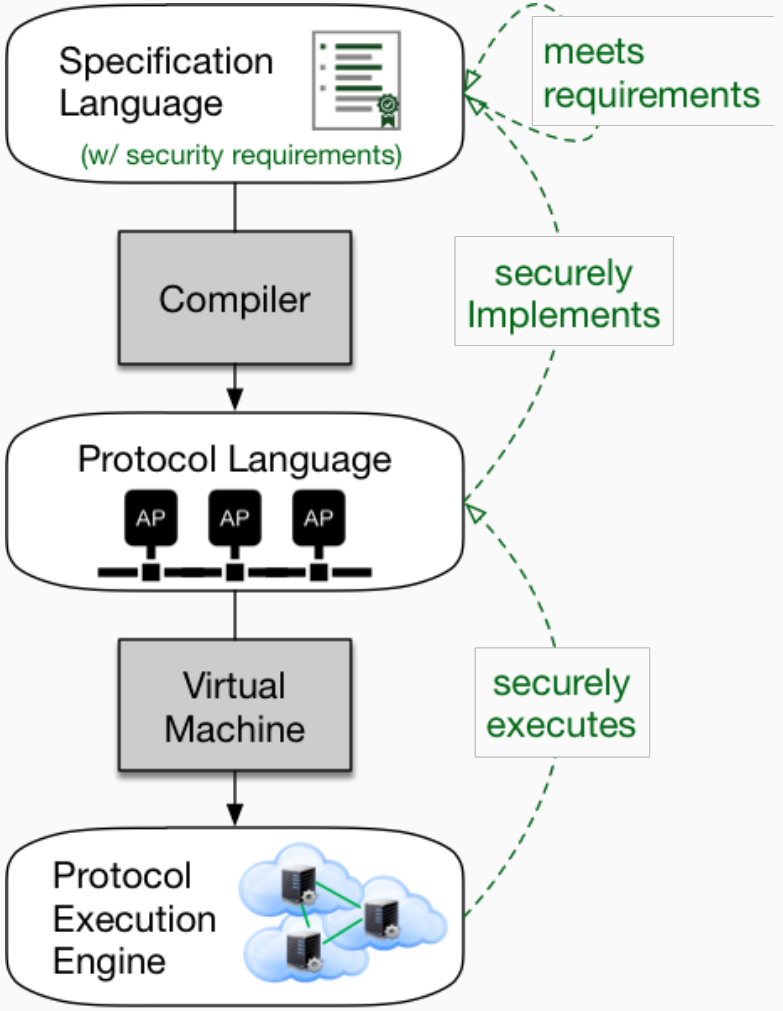
- a programmer: not obvious... all comparisons?

```
forall i; 0<=i<size(xs) && 0<=j<size(xs) ==> public(xs[i] < xs[j])
```

- a MPC expert: nothing! (with suitable preprocessing)

# This paper: A Leakage-Aware MPC Software Stack

- provide early and end-to-end security guarantees for MPC programs
- this presentation (passive w/ leakage)
  - language-based techniques
    - specify security policies
    - automatically check security policies
    - prove secure compilation
  - cryptographic techniques
    - protocol execution
  - language-based security $\Rightarrow$ cryptographic security



Motivation

High-level Language

Low-level Language

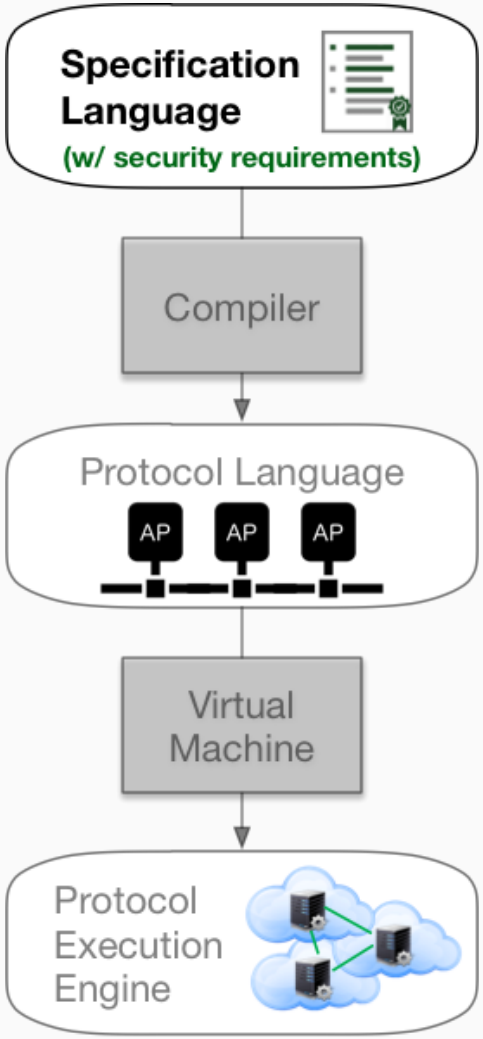
Compilation

Cryptography

Optimization

Tool

Conclusions



# High-level Language

- we adopt SecreC, a C++-like language used for writing MPC applications in the Sharemind framework
- formally, a WHILE language with arrays, declassification and public/secret primitive operations
- standard information-flow type system (public $\sqsubseteq$ secret) that enforces public-control flow
- semantics gives meaning to a TTP computing directly over the data
- small-step semantics, instrumented with leakage

$$\langle p, m \rangle \rightarrow_l \langle p', m' \rangle$$
$$\langle p, m \rangle \Downarrow_l m'$$

**Remember later**

We will assume that all programs are safe

# High-level Security

- program $p$ is secure for $\Phi$ (non-interference)

$$\begin{pmatrix} \langle p, x_1 \rangle \Downarrow_{l_1} y_1 \\ \langle p, x_2 \rangle \Downarrow_{l_2} y_2 \end{pmatrix} \Rightarrow \Phi(x_1, x_2) \Rightarrow l_1 = l_2$$

- relational leakage specification

$$\Phi_\ell(x, y) \triangleq \ell(x) = \ell(y)$$

- relational security Hoare logic $\{\Phi\}\, p\, \{\Psi\}$

$$\begin{pmatrix} \langle p, x_1 \rangle \Downarrow_{l_1} y_1 \\ \langle p, x_2 \rangle \Downarrow_{l_2} y_2 \end{pmatrix} \Rightarrow \Phi(x_1, x_2) \Rightarrow \Psi(y_1, y_2) \wedge l_1 = l_2$$

- compositional reasoning about pairs of executions of the same program running in lockstep.

**Remember later**

Can be efficiently verified using self-composition techniques

Motivation

High-level Language

Low-level Language

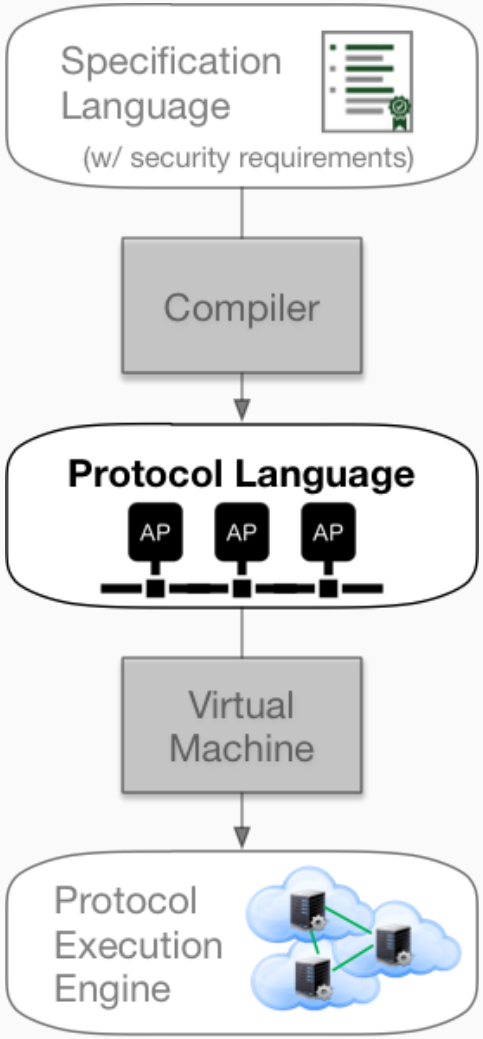
Compilation

Cryptography

Optimization

Tool

Conclusions



# Low-level Language

- low-level semantics runs a program as a distributed MPC protocol
- each party keeps a local state of additive shares

$$M = (M_1, M_2, \ldots, M_n)$$

- secret-shared encoding of public values (no communication)

$$v = (v, v, -v, v, v, -v \ldots)$$

$$v = v + v - v + v + v - v + \ldots$$

- local evaluation rules (no communication, asynchronous execution)

$$\langle p, \mathrm{M}_i \rangle \Rrightarrow \langle p', \mathrm{M}'_i \rangle$$

- global evaluation rules for declassify and secure operations (secure communication, synchronous execution)

$$\langle p, \mathrm{M} \rangle \Rrightarrow_{t,c} \langle p, \mathrm{M}' \rangle$$

# Low-level Security

- protocol $\pi$ is correct for program $p$

$$\langle p, \mathrm{Unshare}(\bar{x})\rangle \Downarrow \mathrm{Unshare}(\bar{y}) \Rightarrow \langle \pi, \bar{x}\rangle \Downarrow_{t,c} \bar{y}$$

- protocol $\pi$ is secure for $\Phi$ (for party $i$) (non-interference)

$$x = \mathrm{Unshare}(\bar{x}) \wedge x' = \mathrm{Unshare}(\bar{x}') \Rightarrow$$
$$\Phi(x, x') \wedge \begin{pmatrix} \langle \pi, \bar{x}\rangle \Downarrow_{t,c} \bar{y} \\ \langle \pi, \bar{x'}\rangle \Downarrow_{t',c'} \bar{y'} \end{pmatrix} \Rightarrow (t_i, c_i) = (t_i', c_i')$$

**Remember later**

Leakage relation over (unshared) values.

Motivation

High-level Language

Low-level Language

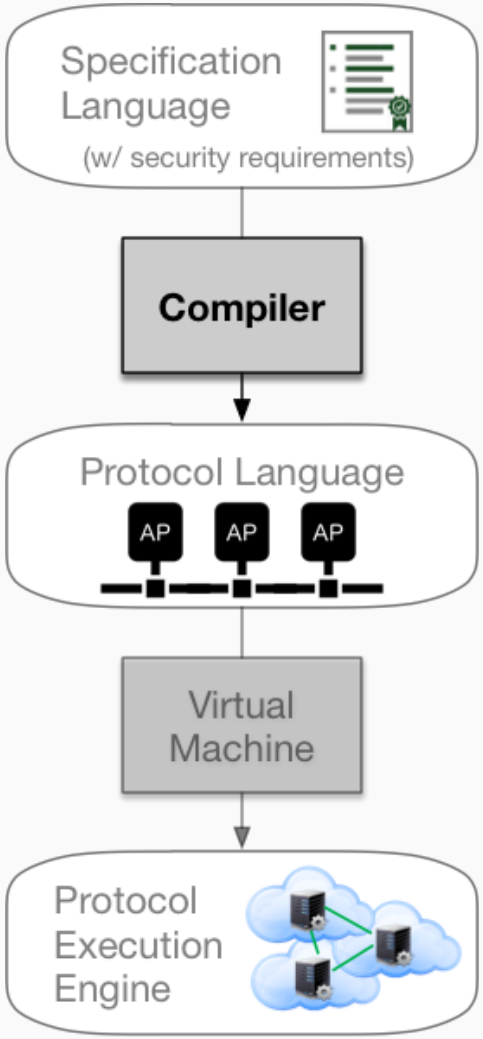
Compilation

Cryptography

Optimization

Tool

Conclusions



# Secure Compilation

- compile a program $p$ into a composite protocol $\pi_p$
- $\pi_p$ = sequence of $\pi_{\text{declassify}}$ and $\pi_{\text{sop}}$

**Secure Compilation**

Let $p$ be a well-typed and $\Phi$-secure program. Then we have that protocol $\pi_p$ is correct for $p$ and secure for $\Phi$.

- *proof sketch*
  - $p$ is well-typed $\Rightarrow$ no secret values in public computations
  - high-level control flow = low-level control flow
  - $\pi_p$ is synchronously executed
  - compositional notions of low-level correctness and security
    - simple proofs by non-interference

Motivation

High-level Language

Low-level Language

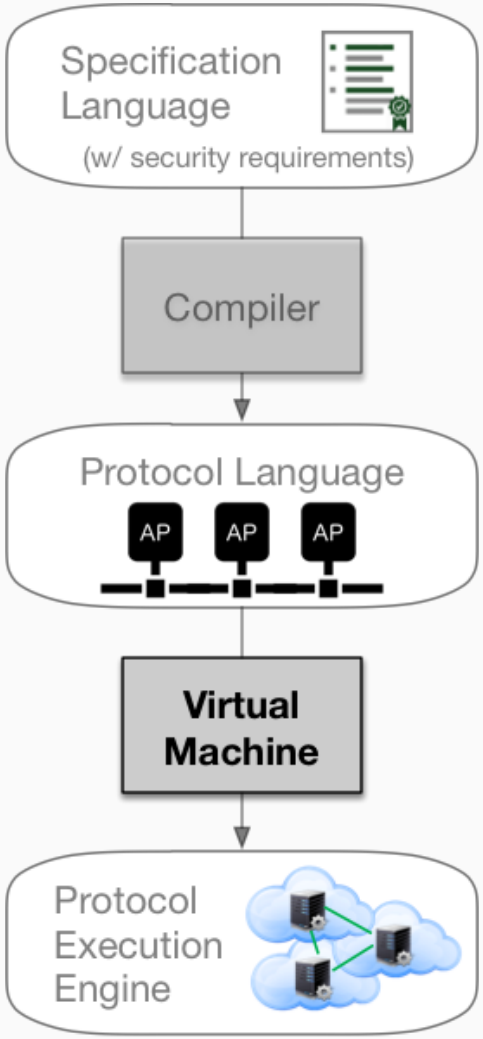
Compilation

Cryptography

Optimization

Tool

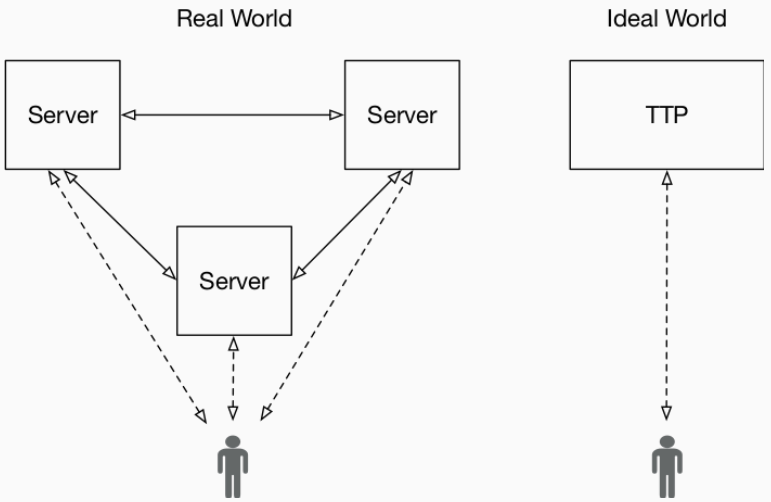
Conclusions



# Cryptography – Real world vs Ideal world

- in the real world, $\mathcal{A}$ interacts with three participants, executing the MPC protocol
- in the ideal world, $\mathcal{A}$ will interact with a trusted party, ideally executing the protocol
- cryptographic security states that the views of $\mathcal{A}$ should be indistinguishable, i.e.

$$\mathsf{Real}_{\mathcal{A}} \equiv \mathsf{Ideal}_{\mathcal{A}}$$




| $\textbf{game } \mathbf{Real}_{\Pi,\mathcal{A}}():$ | $\textbf{game } \mathbf{Ideal}_{\mathcal{F}\langle p,\ell\rangle,\mathcal{S},\mathcal{A}}():$ |
|---|---|
| $(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$ | $(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$ |
| | $x \leftarrow \mathrm{Unshare}(\bar{x})$ |
| | $y \leftarrow p(x)$ |
| | $l \leftarrow \ell(x)$ |
| $(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$ | $(y_i, t, c) \twoheadleftarrow \mathcal{S}(i, l, x_i)$ |
| $y \leftarrow \mathrm{Unshare}(\bar{y})$ | |
| Return $\mathcal{A}_2(\bar{x}, y, y_i, t_i, c_i, \mathsf{st})$ | Return $\mathcal{A}_2(\bar{x}, y, y_i, t_i, c_i, \mathsf{st})$ |

# Cryptography – From language-based security

## Cryptographic Security

correctness $\wedge$ security $\Rightarrow$ crypto security

- *proof sketch*
  - correctness $\Rightarrow$ we can replace a real protocol execution for its corresponding ideal program
  - security $\Rightarrow$ we can construct a simulator that receives the leakage to construct input shares; it can then run the protocol to produce traces and coins that are indistinguishable from the real ones

| $\underline{\textbf{game Real}_{\Pi,\mathcal{A}}()}$: | $\underline{\textbf{game Ideal}_{\mathcal{F}\langle p,\ell\rangle,\mathcal{S},\mathcal{A}}()}$: |
|---|---|
| $(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$ | $(\bar{x}, i, \mathsf{st}) \twoheadleftarrow \mathcal{A}_1()$ |
| | $x \leftarrow \mathrm{Unshare}(\bar{x})$ |
| | $y \leftarrow p(x)$ |
| $(\bar{y}, t, c) \twoheadleftarrow \Pi(\bar{x})$ | $l \leftarrow \ell(x)$ |
| $y \leftarrow \mathrm{Unshare}(\bar{y})$ | $(y_i, t, c) \twoheadleftarrow \mathcal{S}(i, l, x_i)$ |
| Return $\mathcal{A}_2(\bar{x}, y, y_i, t_i, c_i, \mathsf{st})$ | Return $\mathcal{A}_2(\bar{x}, y, y_i, t_i, c_i, \mathsf{st})$ |

Motivation

High-level Language

Low-level Language

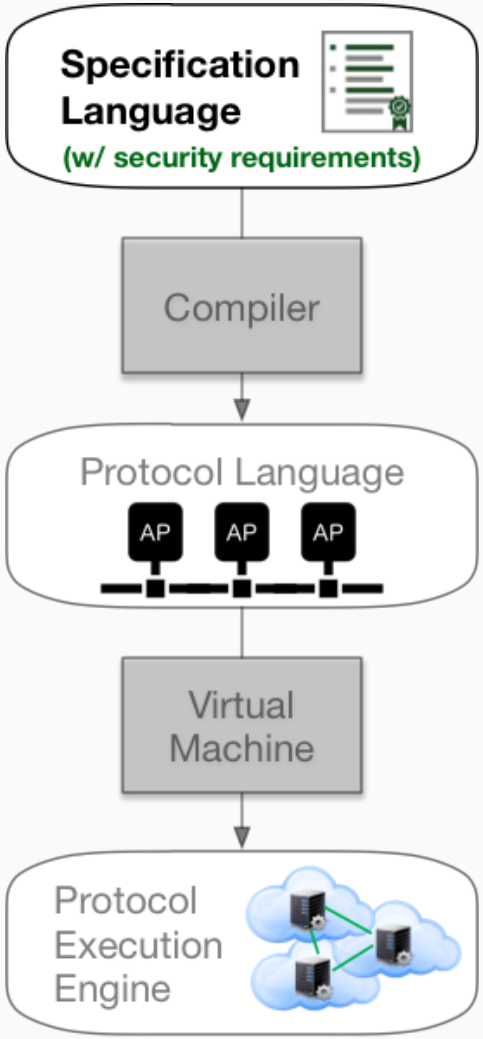
Compilation

Cryptography

Optimization

Tool

Conclusions



# Leakage Cancelling – Example

1. write a (more efficient) program that leaks more than desired (e.g. all comparisons)

```
secret maximum (secret xs) {
    secret m = xs[0];
    for (public i=1; i<size(xs); i+=1)
        if declassify(xs[i]>m) m = xs[i];
    return m; }
```

2. cancel this leakage with an (efficient) probabilistic preprocessing operation (e.g. oblivious shuffle)

```
secret auction (secret xs)
{ return maximum(shuffle(xs)); }
```

**Intuition**

Applying a random permutation to the input makes the sequence of comparisons look random, and useless to an attacker that does not know which permutation was applied (assuming that all elements are distinct).

# Leakage Cancelling – Formally

- lift security to probabilistic programs

$$\begin{pmatrix} \langle p, x_1 \rangle \Downarrow_{\tilde{l_1}} \tilde{y_1} \\ \langle p, x_2 \rangle \Downarrow_{\tilde{l_2}} \tilde{y_2} \end{pmatrix} \Rightarrow \Phi(x_1, x_2) \Rightarrow \tilde{l_1} = \tilde{l_2}$$

- a probabilistic program $p_0$ is a correct preprocessing for a deterministic program $p$

$$\langle p, x \rangle \Downarrow_l y \Rightarrow \langle p_0; p, x \rangle \Downarrow_{l'} \tilde{y} \Rightarrow \Longrightarrow \tilde{y} = \mathbf{1}_y$$

- a $\Phi$-secure program $p_0$ with deterministic leakage is a secure preprocessing for a $\Psi$-secure program $p$ (non-interference)

$$\begin{pmatrix} \langle p_0, x_1 \rangle \Downarrow_{l_1} \tilde{y_1} \\ \langle p_0, x_2 \rangle \Downarrow_{l_2} \tilde{y_2} \end{pmatrix} \Rightarrow \Phi(x_1, x_2) \Rightarrow \tilde{\Psi}(\tilde{y_1}, \tilde{y_2})$$

$$\tilde{\Psi}(\tilde{y_1}, \tilde{y_2}) \triangleq \forall y.\, \mathrm{Pr}_{y_1 \leftarrow \tilde{y_1}}[\Psi(y_1, y)] = \mathrm{Pr}_{y_2 \leftarrow \tilde{y_2}}[\Psi(y_2, y)]$$

Motivation

High-level Language

Low-level Language

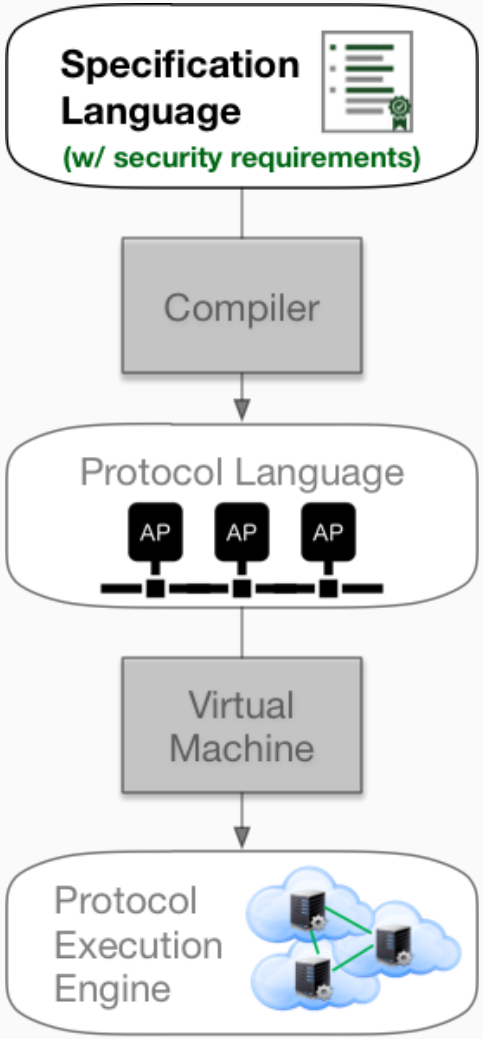
Compilation

Cryptography

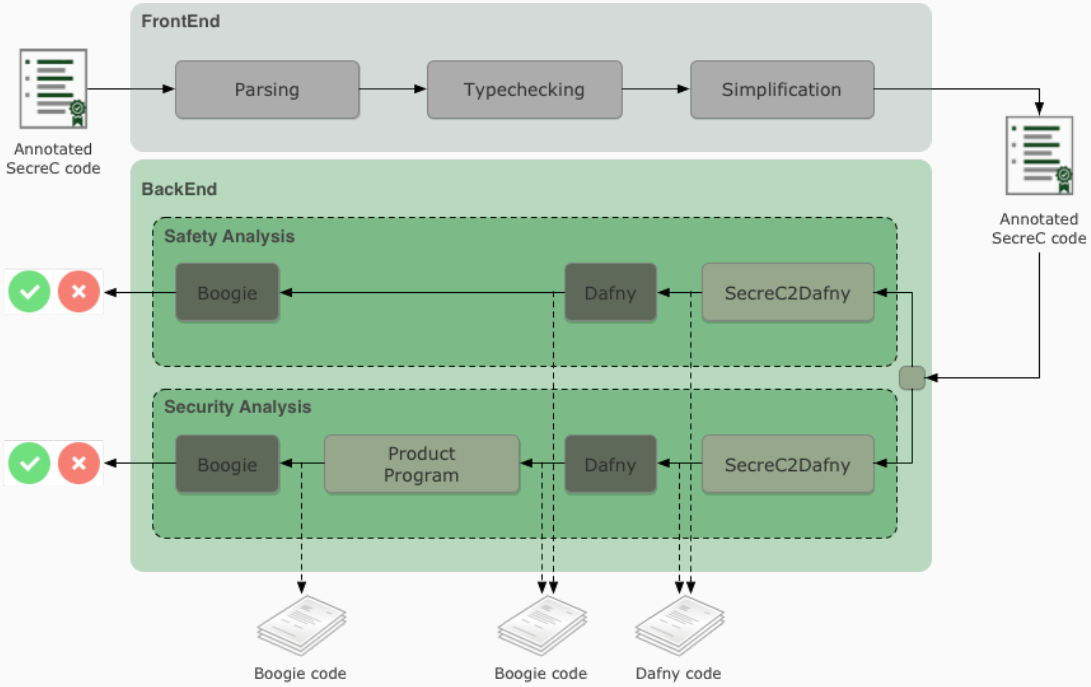
Optimization

Tool

Conclusions



# Implementation - SecreC Verification Tool




- relies on the Dafny-Boogie verification toolchain:
  - safety (for cryptographic security): standard deductive verification
  - security: product programs
- currently: deterministic programs, no support for leakage cancelling

**Development**

https://github.com/haslab/SecreC

# Experiments

- leaky SecreC programs from the Sharemind SDK
  - application server for computing over encrypted data
  - developed by Cybernetica
  - `https://github.com/sharemind-sdk/secrec`

| SecreC | LOC | Leakage (Automated*) | Cancelling (Manual) |
|---|---|---|---|
| quick-sort | 101 | all comparisons | shuffle; leakage $= \emptyset$ |
| radix-sort | 135 | row permutation | shuffle; leakage $= \emptyset$ |
| gaussian | 178 | row permutation | shuffle; leakage $= \emptyset$ |
| $k$-apriori | 414 | frequent itemsets up to $k$ | leakage $=$ output |

* automated verification requires procedure and loop annotations

# Conclusions

- work focus: language-based security treatment for MPC stack
- challenges:
  - programming languages
  - secure compilation
  - cryptographic realizations
- final remarks:
  - possible to achieve secure evaluation for leakage-aware language
  - probabilistic non-interference *vs*. cryptographic security
  - interesting combination of PL and Crypto tools/techniques