ZK-SecreC: A Domain-Specific Language for Zero-Knowledge Proofs

Dan Bogdanov, Joosep Jääger, Peeter Laud, **Härmel Nestra**, Martin Pettai, Jaak Randmets, Ville Sokk, Kert Tali, Sandhra-Mirella Valdma[1]

Joint Estonian-Latvian Theory Days, Riga, 8 May 2022

[1]This research has been funded by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-20-C-0083. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This research has also been supported by European Regional Development Fund through the Estonian Centre of Excellence in ICT Research (EXITE).

Outline

- The notion of zero-knowledge proof
- Specifying zero-knowledge proofs in a high level programming language such as ZK-SecreC
- Type system of ZK-SecreC
- Compilation of ZK-SecreC into arithmetic circuit

Outline

- The notion of zero-knowledge proof
- Specifying zero-knowledge proofs in a high level programming language such as ZK-SecreC
- Type system of ZK-SecreC
- Compilation of ZK-SecreC into arithmetic circuit

Outline

- The notion of zero-knowledge proof
- Specifying zero-knowledge proofs in a high level programming language such as ZK-SecreC
- Type system of ZK-SecreC
- Compilation of ZK-SecreC into arithmetic circuit

Outline

- The notion of zero-knowledge proof
- Specifying zero-knowledge proofs in a high level programming language such as ZK-SecreC
- Type system of ZK-SecreC
- Compilation of ZK-SecreC into arithmetic circuit

Outline

- The notion of zero-knowledge proof
- Specifying zero-knowledge proofs in a high level programming language such as ZK-SecreC
- Type system of ZK-SecreC
- Compilation of ZK-SecreC into arithmetic circuit

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Zero-Knowledge Proof

- Participants:
  - Two parties called Prover and Verifier
  - A predicate whose specification is known to both parties
  - Possibly a computation environment that both parties trust
- Act of communication:
  - Prover convinces Verifier of knowing an argument that makes the predicate true
  - The information obtained by Verifier does not facilitate finding this argument oneself

Motivating examples

- Prover convinces Verifier of knowing the right password for logging into a server.
- Prover convinces a color-blind Verifier of balls of different colors existing in a pool of otherwise identical balls.
- Prover convinces Verifier of knowing a non-trivial factor of a large integer.
- ...

Motivating examples

- Prover convinces Verifier of knowing the right password for logging into a server.
- Prover convinces a color-blind Verifier of balls of different colors existing in a pool of otherwise identical balls.
- Prover convinces Verifier of knowing a non-trivial factor of a large integer.
- ...

Motivating examples

- Prover convinces Verifier of knowing the right password for logging into a server.
- Prover convinces a color-blind Verifier of balls of different colors existing in a pool of otherwise identical balls.
- Prover convinces Verifier of knowing a non-trivial factor of a large integer.
- ...

Motivating examples

- Prover convinces Verifier of knowing the right password for logging into a server.
- Prover convinces a color-blind Verifier of balls of different colors existing in a pool of otherwise identical balls.
- Prover convinces Verifier of knowing a non-trivial factor of a large integer.
- . . .

Motivating examples

- Prover convinces Verifier of knowing the right password for logging into a server.
- Prover convinces a color-blind Verifier of balls of different colors existing in a pool of otherwise identical balls.
- Prover convinces Verifier of knowing a non-trivial factor of a large integer.
- ...

Example: Prover can distinguish identical balls:

1. Prover picks two balls of distinct colors from the pool;
2. Verifier arranges the balls chosen by Prover into some order;
3. Prover closes eyes and Verifier either switches the balls or not;
4. Prover opens eyes and tells whether Verifier switched the balls or not;
5. While Verifier is still not convinced, go to step 3;
6. Prover puts the balls back to the pool and shuffles.

Example: Prover can distinguish identical balls:

1. Prover picks two balls of distinct colors from the pool;
2. Verifier arranges the balls chosen by Prover into some order;
3. Prover closes eyes and Verifier either switches the balls or not;
4. Prover opens eyes and tells whether Verifier switched the balls or not;
5. While Verifier is still not convinced, go to step 3;
6. Prover puts the balls back to the pool and shuffles.

Example: Prover can distinguish identical balls:

1. Prover picks two balls of distinct colors from the pool;
2. Verifier arranges the balls chosen by Prover into some order;
3. Prover closes eyes and Verifier either switches the balls or not;
4. Prover opens eyes and tells whether Verifier switched the balls or not;
5. While Verifier is still not convinced, go to step 3;
6. Prover puts the balls back to the pool and shuffles.

Example: Prover can distinguish identical balls:

1. Prover picks two balls of distinct colors from the pool;
2. Verifier arranges the balls chosen by Prover into some order;
3. Prover closes eyes and Verifier either switches the balls or not;
4. Prover opens eyes and tells whether Verifier switched the balls or not;
5. While Verifier is still not convinced, go to step 3;
6. Prover puts the balls back to the pool and shuffles.

Example: Prover can distinguish identical balls:

1. Prover picks two balls of distinct colors from the pool;
2. Verifier arranges the balls chosen by Prover into some order;
3. Prover closes eyes and Verifier either switches the balls or not;
4. Prover opens eyes and tells whether Verifier switched the balls or not;
5. While Verifier is still not convinced, go to step 3;
6. Prover puts the balls back to the pool and shuffles.

Example: Prover can distinguish identical balls:

1. Prover picks two balls of distinct colors from the pool;
2. Verifier arranges the balls chosen by Prover into some order;
3. Prover closes eyes and Verifier either switches the balls or not;
4. Prover opens eyes and tells whether Verifier switched the balls or not;
5. While Verifier is still not convinced, go to step 3;
6. Prover puts the balls back to the pool and shuffles.

Example: Prover can distinguish identical balls:

1. Prover picks two balls of distinct colors from the pool;
2. Verifier arranges the balls chosen by Prover into some order;
3. Prover closes eyes and Verifier either switches the balls or not;
4. Prover opens eyes and tells whether Verifier switched the balls or not;
5. While Verifier is still not convinced, go to step 3;
6. Prover puts the balls back to the pool and shuffles.

ZK-SecreC

- A domain-specific programming language under development in Cybernetica AS
- Targetted to specifying zero-knowledge proofs
- Privacy guaranteed (i.e., information leaks excluded) by strong type system
- Compilation into arithmetic circuit:
  - At least indexed input, modular addition and multiplication and assertion of being zero supported
  - A universal common set of primitive operations suitable as input for various ZK proof systems

ZK-SecreC

- A domain-specific programming language under development in Cybernetica AS
- Targetted to specifying zero-knowledge proofs
- Privacy guaranteed (i.e., information leaks excluded) by strong type system
- Compilation into arithmetic circuit:
  - At least indexed input, modular addition and multiplication and assertion of being zero supported
  - A universal common set of primitive operations suitable as input for various ZK proof systems

ZK-SecreC

- A domain-specific programming language under development in Cybernetica AS
- Targetted to specifying zero-knowledge proofs
- Privacy guaranteed (i.e., information leaks excluded) by strong type system
- Compilation into arithmetic circuit:
  - At least indexed input, modular addition and multiplication and assertion of being zero supported
  - A universal common set of primitive operations suitable as input for various ZK proof systems

ZK-SecreC

- A domain-specific programming language under development in Cybernetica AS
- Targetted to specifying zero-knowledge proofs
- Privacy guaranteed (i.e., information leaks excluded) by strong type system
- Compilation into arithmetic circuit:
  - At least indexed input, modular addition and multiplication and assertion of being zero supported
  - A universal common set of primitive operations suitable as input for various ZK proof systems

ZK-SecreC

- A domain-specific programming language under development in Cybernetica AS
- Targetted to specifying zero-knowledge proofs
- Privacy guaranteed (i.e., information leaks excluded) by strong type system
- Compilation into arithmetic circuit:
  - At least indexed input, modular addition and multiplication and assertion of being zero supported
  - A universal common set of primitive operations suitable as input for various ZK proof systems

ZK-SecreC

- A domain-specific programming language under development in Cybernetica AS
- Targetted to specifying zero-knowledge proofs
- Privacy guaranteed (i.e., information leaks excluded) by strong type system
- Compilation into arithmetic circuit:
  - At least indexed input, modular addition and multiplication and assertion of being zero supported
  - A universal common set of primitive operations suitable as input for various ZK proof systems

ZK-SecreC

- A domain-specific programming language under development in Cybernetica AS
- Targetted to specifying zero-knowledge proofs
- Privacy guaranteed (i.e., information leaks excluded) by strong type system
- Compilation into arithmetic circuit:
  - At least indexed input, modular addition and multiplication and assertion of being zero supported
  - A universal common set of primitive operations suitable as input for various ZK proof systems

Example: Prover knows a non-trivial factor

```
fn main() {
    let fbw : uint $pre @public = get_public("fixed_bit_width");
    let z : uint[N] $post @verifier = wire { get_instance("z") };
    let x : uint[N] $post @prover = wire { get_witness("x") };
    let y = wire { z as $pre as @prover / x as $pre };
    assert_zero(x * y - (z as @prover));
    assert(less_than(x, z, fbw));
    assert(less_than(y, z, fbw));
}

fn less_than[@D1, @D2, @D](x : uint[N] $post @D1, y : uint[N] $post @D2, fbw : uint $pre
    @public) -> bool[N] $post @D
    where @D1 <= @D, @D2 <= @D {
    let xb = bitextract(x as @D, fbw);
    let yb = bitextract(y as @D, fbw);
    // lexicographic comparison of lists of bits xb and yb omitted
}
```

Example: Prover knows a non-trivial factor

```
fn main() {
    let fbw : uint $pre @public = get_public("fixed_bit_width");
    let z : uint[N] $post @verifier = wire { get_instance("z") };
    let x : uint[N] $post @prover = wire { get_witness("x") };
    let y = wire { z as $pre as @prover / x as $pre };
    assert_zero(x * y - (z as @prover));
    assert(less_than(x, z, fbw));
    assert(less_than(y, z, fbw));
}

fn less_than[@D1, @D2, @D](x : uint[N] $post @D1, y : uint[N] $post @D2, fbw : uint $pre
    @public) -> bool[N] $post @D
    where @D1 <= @D, @D2 <= @D {
    let xb = bitextract(x as @D, fbw);
    let yb = bitextract(y as @D, fbw);
    // lexicographic comparison of lists of bits xb and yb omitted
}
```

Example: Prover knows a non-trivial factor

```
fn main() {
    let fbw : uint $pre @public = get_public("fixed_bit_width");
    let z : uint[N] $post @verifier = wire { get_instance("z") };
    let x : uint[N] $post @prover = wire { get_witness("x") };
    let y = wire { z as $pre as @prover / x as $pre };
    assert_zero(x * y − (z as @prover));
    assert(less_than(x, z, fbw));
    assert(less_than(y, z, fbw));
}

fn less_than[@D1, @D2, @D](x : uint[N] $post @D1, y : uint[N] $post @D2, fbw : uint $pre
      @public) −> bool[N] $post @D
    where @D1 <= @D, @D2 <= @D {
    let xb = bitextract(x as @D, fbw);
    let yb = bitextract(y as @D, fbw);
    // lexicographic comparison of lists of bits xb and yb omitted
}
```

Example: Prover knows a non-trivial factor (ctd.)

```
fn bitextract[@D](x : uint[N] $post @D, fbw : uint $pre @public) -> list[bool[N] $post @D] {
    let xb_pre = bitextract_pre(x as $pre, fbw);
    let xb = for i in 0 .. length(xb_pre) { wire { xb_pre[i] } };
    if (@prover <= @D) { check_bitextract(x, xb); };
    xb
}

fn bitextract_pre[@D](x : uint[N] $pre @D, fbw : uint $pre @public) -> list[bool[N] $pre @D] {
    let rec xx = for i in 0 .. fbw { if (i == 0) { x } else { xx[i - 1] / 2 } };
    for i in 0 .. fbw { xx[i] % 2 == 1 }
}

fn check_bitextract[@D](x : uint[N] $post @D, xb : list[bool[N] $post @D]) {
    let mut s = xb[length(xb) - 1] as uint[N]; // variable s is mutable
    for i in 0 .. length(xb) - 1 { s = 2 * s + xb[length(xb) - i - 2] as uint[N]; };
    assert_zero(x - s);
}
```

Example: Prover knows a non-trivial factor (ctd.)

```
fn bitextract[@D](x : uint[N] $post @D, fbw : uint $pre @public) -> list[bool[N] $post @D] {
    let xb_pre = bitextract_pre(x as $pre, fbw);
    let xb = for i in 0 .. length(xb_pre) { wire { xb_pre[i] } };
    if (@prover <= @D) { check_bitextract(x, xb); };
    xb
}

fn bitextract_pre[@D](x : uint[N] $pre @D, fbw : uint $pre @public) -> list[bool[N] $pre @D] {
    let rec xx = for i in 0 .. fbw { if (i == 0) { x } else { xx[i - 1] / 2 } };
    for i in 0 .. fbw { xx[i] % 2 == 1 }
}

fn check_bitextract[@D](x : uint[N] $post @D, xb : list[bool[N] $post @D]) {
    let mut s = xb[length(xb) - 1] as uint[N]; // variable s is mutable
    for i in 0 .. length(xb) - 1 { s = 2 * s + xb[length(xb) - i - 2] as uint[N]; };
    assert_zero(x - s);
}
```

Example: Prover knows a non-trivial factor (ctd.)

```
fn bitextract[@D](x : uint[N] $post @D, fbw : uint $pre @public) -> list[bool[N] $post @D] {
    let xb_pre = bitextract_pre(x as $pre, fbw);
    let xb = for i in 0 .. length(xb_pre) { wire { xb_pre[i] } };
    if (@prover <= @D) { check_bitextract(x, xb); };
    xb
}

fn bitextract_pre[@D](x : uint[N] $pre @D, fbw : uint $pre @public) -> list[bool[N] $pre @D] {
    let rec xx = for i in 0 .. fbw { if (i == 0) { x } else { xx[i - 1] / 2 } };
    for i in 0 .. fbw { xx[i] % 2 == 1 }
}

fn check_bitextract[@D](x : uint[N] $post @D, xb : list[bool[N] $post @D]) {
    let mut s = xb[length(xb) - 1] as uint[N]; // variable s is mutable
    for i in 0 .. length(xb) - 1 { s = 2 * s + xb[length(xb) - i - 2] as uint[N]; };
    assert_zero(x - s);
}
```

Example: Prover knows a non-trivial factor (ctd.)

```
fn bitextract[@D](x : uint[N] $post @D, fbw : uint $pre @public) -> list[bool[N] $post @D] {
    let xb_pre = bitextract_pre(x as $pre, fbw);
    let xb = for i in 0 .. length(xb_pre) { wire { xb_pre[i] } };
    if (@prover <= @D) { check_bitextract(x, xb); };
    xb
}

fn bitextract_pre[@D](x : uint[N] $pre @D, fbw : uint $pre @public) -> list[bool[N] $pre @D] {
    let rec xx = for i in 0 .. fbw { if (i == 0) { x } else { xx[i - 1] / 2 } };
    for i in 0 .. fbw { xx[i] % 2 == 1 }
}

fn check_bitextract[@D](x : uint[N] $post @D, xb : list[bool[N] $post @D]) {
    let mut s = xb[length(xb) - 1] as uint[N]; // variable s is mutable
    for i in 0 .. length(xb) - 1 { s = 2 * s + xb[length(xb) - i - 2] as uint[N]; };
    assert_zero(x - s);
}
```

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, . . .) or compound (list, . . .).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Type system

- Qualified types consist of a data type, a stage and a domain.
- Domain can be **@public**, **@verifier** or **@prover**.
  - Specifies the party that knows the particular data.
  - **@public** means that the data is available to the compiler.
- Stage can be **$pre** or **$post**.
  - Specifies the location of computation of the particular data (local or circuit).
- Data type specifies the nature of the data.
  - Can be primitive (modular integer, boolean, ...) or compound (list, ...).
  - Qualifiers of a list type need not coincide with qualifiers of its element type.

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** <: **@verifier** <: **@prover**
  - Stages also ordered by growing unreliability: **$post** <: **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** <: **@verifier** <: **@prover**
  - Stages also ordered by growing unreliability: **$post** <: **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Features

- Subtyping:
  - Domains ordered by growing privacy: **@public** $<:$ **@verifier** $<:$ **@prover**
  - Stages also ordered by growing unreliability: **$post** $<:$ **$pre**
  - Expressions castable into a larger type but not into a smaller one
- Type parameters:
  - Both qualified types and their constituents representable by parameters
  - Domain polymorphism and stage polymorphism
- Type inference:
  - Types required only in function signatures, others inferred by the compiler
  - Domains and stages required only if they cannot be inferred

Effects

- Keeping track of domains where computing an expression/statement can produce effects in:
  - Assignment to a variable whose domain is d causing an effect in domain d
  - All operations in **$post** causing an effect in domain **@public** since contributing to the output circuit
- Typing assertions of the form $\Gamma \vdash e : t \,!\, D$ where Γ is a type environment, e is an expression, t is a qualified type, and D is an upward closed set of domains

Effects

- Keeping track of domains where computing an expression/statement can produce effects in:
  - Assignment to a variable whose domain is d causing an effect in domain d
  - All operations in **$post** causing an effect in domain **@public** since contributing to the output circuit
- Typing assertions of the form $\Gamma \vdash e : t \,!\, D$ where Γ is a type environment, e is an expression, t is a qualified type, and D is an upward closed set of domains

Effects

- Keeping track of domains where computing an expression/statement can produce effects in:
  - Assignment to a variable whose domain is d causing an effect in domain d
  - All operations in **$post** causing an effect in domain **@public** since contributing to the output circuit
- Typing assertions of the form $\Gamma \vdash e : t \,!\, D$ where Γ is a type environment, e is an expression, t is a qualified type, and D is an upward closed set of domains

Effects

- Keeping track of domains where computing an expression/statement can produce effects in:
  - Assignment to a variable whose domain is d causing an effect in domain d
  - All operations in **$post** causing an effect in domain **@public** since contributing to the output circuit
- Typing assertions of the form $\Gamma \vdash e : t \,!\, D$ where Γ is a type environment, e is an expression, t is a qualified type, and D is an upward closed set of domains

Effects

- Keeping track of domains where computing an expression/statement can produce effects in:
  - Assignment to a variable whose domain is d causing an effect in domain d
  - All operations in **$post** causing an effect in domain **@public** since contributing to the output circuit
- Typing assertions of the form $\Gamma \vdash e : t \,!\, D$ where Γ is a type environment, e is an expression, t is a qualified type, and D is an upward closed set of domains

Upward closed sets and the $\langle\cdot\rangle$ notation

Definition

A set of domains is upward closed iff, along with each element domain, the set contains all larger domains.

- Four upward closed sets: $\varnothing \subset \langle \texttt{@prover} \rangle \subset \langle \texttt{@verifier} \rangle \subset \langle \texttt{@public} \rangle$, where $\langle d \rangle$ denotes the set consisting of domain d and all larger domains
- The notation extended to stages by $\langle \texttt{\$pre} \rangle = \varnothing$ and $\langle \texttt{\$post} \rangle = \langle \texttt{@public} \rangle$

Upward closed sets and the $\langle\cdot\rangle$ notation

Definition

A set of domains is upward closed iff, along with each element domain, the set contains all larger domains.

- Four upward closed sets: $\varnothing \subset \langle \textbf{@prover} \rangle \subset \langle \textbf{@verifier} \rangle \subset \langle \textbf{@public} \rangle$, where $\langle d \rangle$ denotes the set consisting of domain d and all larger domains
- The notation extended to stages by $\langle \$\textbf{pre} \rangle = \varnothing$ and $\langle \$\textbf{post} \rangle = \langle \textbf{@public} \rangle$

Upward closed sets and the $\langle\cdot\rangle$ notation

Definition

A set of domains is upward closed iff, along with each element domain, the set contains all larger domains.

- Four upward closed sets: $\varnothing \subset \langle$**@prover**$\rangle \subset \langle$**@verifier**$\rangle \subset \langle$**@public**$\rangle$, where $\langle d \rangle$ denotes the set consisting of domain d and all larger domains
- The notation extended to stages by $\langle$**$pre**$\rangle = \varnothing$ and $\langle$**$post**$\rangle = \langle$**@public**$\rangle$

Examples of type rules

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ \langle s \rangle} \qquad \frac{(x : t\ s\ d) \in \Gamma}{\Gamma \vdash x : t\ s\ d\ !\ \varnothing}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ D_1 \quad \Gamma \vdash e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ D_2}{\Gamma \vdash e_1 + e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ \langle s \rangle \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{uint}\ \mathbf{\$pre}\ d'\ !\ D_i\ (i = 1, 2) \quad (x : \mathbf{uint}\ \mathbf{\$pre}\ d'), \Gamma \vdash e_3 : t\ s\ d\ !\ D_3 \quad \langle d' \rangle \supseteq \langle s \rangle \cup \langle d \rangle \cup D_3}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ ..\ e_2\ \{\ e_3\ \} : \mathbf{list}[t\ s\ d]\ \mathbf{\$pre}\ d'\ !\ D_1 \cup D_2 \cup D_3}$$

$$\frac{\Gamma \vdash e : t\ \mathbf{\$pre}\ d\ !\ D \quad t \in \{\mathbf{uint}[\mathrm{N}], \mathbf{bool}[\mathrm{N}]\}}{\Gamma \vdash \mathbf{wire}\ \{\ e\ \} : t\ \mathbf{\$post}\ d\ !\ \langle \mathbf{@public} \rangle} \qquad \frac{\Gamma \vdash e : t\ s\ d\ !\ D \quad \Gamma \vdash l : t\ s\ d\ !\ D'}{\Gamma \vdash l = e : ()\ \mathbf{\$pre}\ \mathbf{@public}\ !\ \langle s \rangle \cup \langle d \rangle \cup D \cup D'}$$

Examples of type rules

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle} \qquad \frac{(x : t\ s\ d) \in \Gamma}{\Gamma \vdash x : t\ s\ d\,!\,\varnothing}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_1 \quad \Gamma \vdash e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_2}{\Gamma \vdash e_1 + e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{uint}\ \mathbf{\$pre}\ d'\,!\,D_i\ (i = 1, 2) \quad (x : \mathbf{uint}\ \mathbf{\$pre}\ d'), \Gamma \vdash e_3 : t\ s\ d\,!\,D_3 \quad \langle d'\rangle \supseteq \langle s\rangle \cup \langle d\rangle \cup D_3}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ ..\ e_2\ \{\ e_3\ \} : \mathbf{list}[t\ s\ d]\ \mathbf{\$pre}\ d'\,!\,D_1 \cup D_2 \cup D_3}$$

$$\frac{\Gamma \vdash e : t\ \mathbf{\$pre}\ d\,!\,D \quad t \in \{\mathbf{uint}[\mathrm{N}], \mathbf{bool}[\mathrm{N}]\}}{\Gamma \vdash \mathbf{wire}\ \{\ e\ \} : t\ \mathbf{\$post}\ d\,!\,\langle \mathbf{@public}\rangle} \qquad \frac{\Gamma \vdash e : t\ s\ d\,!\,D \quad \Gamma \vdash l : t\ s\ d\,!\,D'}{\Gamma \vdash l = e : ()\ \mathbf{\$pre}\ \mathbf{@public}\,!\,\langle s\rangle \cup \langle d\rangle \cup D \cup D'}$$

Examples of type rules

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle} \qquad \frac{(x : t\ s\ d) \in \Gamma}{\Gamma \vdash x : t\ s\ d\,!\,\varnothing}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_1 \quad \Gamma \vdash e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_2}{\Gamma \vdash e_1 + e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{uint}\ \mathbf{\$pre}\ d'\,!\,D_i\ (i = 1, 2) \quad (x : \mathbf{uint}\ \mathbf{\$pre}\ d'), \Gamma \vdash e_3 : t\ s\ d\,!\,D_3 \quad \langle d'\rangle \supseteq \langle s\rangle \cup \langle d\rangle \cup D_3}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ ..\ e_2\ \{\ e_3\ \} : \mathbf{list}[t\ s\ d]\ \mathbf{\$pre}\ d'\,!\,D_1 \cup D_2 \cup D_3}$$

$$\frac{\Gamma \vdash e : t\ \mathbf{\$pre}\ d\,!\,D \quad t \in \{\mathbf{uint}[\mathrm{N}], \mathbf{bool}[\mathrm{N}]\}}{\Gamma \vdash \mathbf{wire}\ \{\ e\ \} : t\ \mathbf{\$post}\ d\,!\,\langle \mathbf{@public}\rangle} \qquad \frac{\Gamma \vdash e : t\ s\ d\,!\,D \quad \Gamma \vdash l : t\ s\ d\,!\,D'}{\Gamma \vdash l = e : ()\ \mathbf{\$pre}\ \mathbf{@public}\,!\,\langle s\rangle \cup \langle d\rangle \cup D \cup D'}$$

Examples of type rules

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle} \qquad \frac{(x : t\ s\ d) \in \Gamma}{\Gamma \vdash x : t\ s\ d\,!\,\varnothing}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_1 \quad \Gamma \vdash e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_2}{\Gamma \vdash e_1 + e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{uint}\ \$\mathbf{pre}\ d'\,!\,D_i\ (i = 1, 2) \quad (x : \mathbf{uint}\ \$\mathbf{pre}\ d'), \Gamma \vdash e_3 : t\ s\ d\,!\,D_3 \quad \langle d'\rangle \supseteq \langle s\rangle \cup \langle d\rangle \cup D_3}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ ..\ e_2\ \{\ e_3\ \} : \mathbf{list}[t\ s\ d]\ \$\mathbf{pre}\ d'\,!\,D_1 \cup D_2 \cup D_3}$$

$$\frac{\Gamma \vdash e : t\ \$\mathbf{pre}\ d\,!\,D \quad t \in \{\mathbf{uint}[\mathrm{N}], \mathbf{bool}[\mathrm{N}]\}}{\Gamma \vdash \mathbf{wire}\ \{\ e\ \} : t\ \$\mathbf{post}\ d\,!\,\langle @\mathbf{public}\rangle} \qquad \frac{\Gamma \vdash e : t\ s\ d\,!\,D \quad \Gamma \vdash l : t\ s\ d\,!\,D'}{\Gamma \vdash l = e : ()\ \$\mathbf{pre}\ @\mathbf{public}\,!\,\langle s\rangle \cup \langle d\rangle \cup D \cup D'}$$

Examples of type rules

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ \langle s \rangle} \qquad \frac{(x : t\ s\ d) \in \Gamma}{\Gamma \vdash x : t\ s\ d\ !\ \varnothing}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ D_1 \quad \Gamma \vdash e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ D_2}{\Gamma \vdash e_1 + e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ \langle s \rangle \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{uint}\ \mathbf{\$pre}\ d'\ !\ D_i\ (i = 1, 2) \quad (x : \mathbf{uint}\ \mathbf{\$pre}\ d'), \Gamma \vdash e_3 : t\ s\ d\ !\ D_3 \quad \langle d' \rangle \supseteq \langle s \rangle \cup \langle d \rangle \cup D_3}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ ..\ e_2\ \{\ e_3\ \} : \mathbf{list}[t\ s\ d]\ \mathbf{\$pre}\ d'\ !\ D_1 \cup D_2 \cup D_3}$$

$$\frac{\Gamma \vdash e : t\ \mathbf{\$pre}\ d\ !\ D \quad t \in \{\mathbf{uint}[\mathrm{N}], \mathbf{bool}[\mathrm{N}]\}}{\Gamma \vdash \mathbf{wire}\ \{\ e\ \} : t\ \mathbf{\$post}\ d\ !\ \langle \mathbf{@public} \rangle} \qquad \frac{\Gamma \vdash e : t\ s\ d\ !\ D \quad \Gamma \vdash l : t\ s\ d\ !\ D'}{\Gamma \vdash l = e : ()\ \mathbf{\$pre}\ \mathbf{@public}\ !\ \langle s \rangle \cup \langle d \rangle \cup D \cup D'}$$

Examples of type rules

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ \langle s \rangle} \qquad \frac{(x : t\ s\ d) \in \Gamma}{\Gamma \vdash x : t\ s\ d\ !\ \varnothing}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ D_1 \quad \Gamma \vdash e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ D_2}{\Gamma \vdash e_1 + e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\ !\ \langle s \rangle \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{uint}\ \mathbf{\$pre}\ d'\ !\ D_i\ (i = 1, 2) \quad (x : \mathbf{uint}\ \mathbf{\$pre}\ d'), \Gamma \vdash e_3 : t\ s\ d\ !\ D_3 \quad \langle d' \rangle \supseteq \langle s \rangle \cup \langle d \rangle \cup D_3}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ ..\ e_2\ \{\ e_3\ \} : \mathbf{list}[t\ s\ d]\ \mathbf{\$pre}\ d'\ !\ D_1 \cup D_2 \cup D_3}$$

$$\frac{\Gamma \vdash e : t\ \mathbf{\$pre}\ d\ !\ D \quad t \in \{\mathbf{uint}[\mathrm{N}], \mathbf{bool}[\mathrm{N}]\}}{\Gamma \vdash \mathbf{wire}\ \{\ e\ \} : t\ \mathbf{\$post}\ d\ !\ \langle \mathbf{@public} \rangle} \qquad \frac{\Gamma \vdash e : t\ s\ d\ !\ D \quad \Gamma \vdash l : t\ s\ d\ !\ D'}{\Gamma \vdash l = e : ()\ \mathbf{\$pre}\ \mathbf{@public}\ !\ \langle s \rangle \cup \langle d \rangle \cup D \cup D'}$$

Examples of type rules

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle} \qquad \frac{(x : t\ s\ d) \in \Gamma}{\Gamma \vdash x : t\ s\ d\,!\,\varnothing}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_1 \quad \Gamma \vdash e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,D_2}{\Gamma \vdash e_1 + e_2 : \mathbf{uint}[\mathrm{N}]\ s\ d\,!\,\langle s\rangle \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{uint}\ \mathbf{\$pre}\ d'\,!\,D_i\ (i = 1, 2) \quad (x : \mathbf{uint}\ \mathbf{\$pre}\ d'), \Gamma \vdash e_3 : t\ s\ d\,!\,D_3 \quad \langle d'\rangle \supseteq \langle s\rangle \cup \langle d\rangle \cup D_3}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ ..\ e_2\ \{\ e_3\ \} : \mathbf{list}[t\ s\ d]\ \mathbf{\$pre}\ d'\,!\,D_1 \cup D_2 \cup D_3}$$

$$\frac{\Gamma \vdash e : t\ \mathbf{\$pre}\ d\,!\,D \quad t \in \{\mathbf{uint}[\mathrm{N}], \mathbf{bool}[\mathrm{N}]\}}{\Gamma \vdash \mathbf{wire}\ \{\ e\ \} : t\ \mathbf{\$post}\ d\,!\,\langle \mathbf{@public}\rangle} \qquad \frac{\Gamma \vdash e : t\ s\ d\,!\,D \quad \Gamma \vdash l : t\ s\ d\,!\,D'}{\Gamma \vdash l = e : ()\ \mathbf{\$pre}\ \mathbf{@public}\,!\,\langle s\rangle \cup \langle d\rangle \cup D \cup D'}$$

Properties

Definition

- Let $q = t\ s\ d$ be a qualified type. We call q well-structured if either t is a primitive type [...] or $s = \$\mathbf{pre}$ and $t = \mathbf{list}[t'\ s'\ d']$ such that $\langle d \rangle \supseteq \langle s' \rangle \cup \langle d' \rangle$ and $t'\ s'\ d'$ is well-structured.
- Call a type environment Γ well-structured if all qualified types occurring in it are well-structured.

Theorem

If $\Gamma \vdash e : q\,!\,D$ *with a well-structured* Γ *then* q *is well-structured.*

Circuits

Definition

An arithmetic circuit over a ring R consists of:

- A direct asyclic graph;
- A partitioning of nodes into input, constant and operation nodes, such that every operation node has exactly two incoming arcs and the other nodes have none;
- An assignment of elements of R to constant nodes, either addition or multiplication of R to every operation node, and either **@prover** or **@verifier** to every input node;
- A subset of nodes specified as output nodes;
- An indexing of all input nodes of each domain by natural numbers.

Evaluation of circuits

Definition

Let $\mathcal{V}$ be the set of all nodes in circuit and $\mathcal{I} \subseteq \mathcal{V}$ the set of all input nodes.

- An assignment $\alpha \in R^{\mathcal{I}}$ of values to the input nodes extends naturally to an assignment $\alpha^\star \in R^{\mathcal{V}}$ to all nodes.
- We say that the circuit accepts input $\alpha \in R^{\mathcal{I}}$, if $\alpha^\star$ assigns 0 to all output nodes.

- Various ZKP techniques applicable for evaluation if R is a finite field of characteristic N (perhaps with additional restrictions on N)

Evaluation of circuits

Definition

Let $\mathcal{V}$ be the set of all nodes in circuit and $\mathcal{I} \subseteq \mathcal{V}$ the set of all input nodes.

- An assignment $\alpha \in R^{\mathcal{I}}$ of values to the input nodes extends naturally to an assignment $\alpha^\star \in R^{\mathcal{V}}$ to all nodes.
- We say that the circuit accepts input $\alpha \in R^{\mathcal{I}}$, if $\alpha^\star$ assigns 0 to all output nodes.

- Various ZKP techniques applicable for evaluation if R is a finite field of characteristic $\mathtt{N}$ (perhaps with additional restrictions on $\mathtt{N}$)

Types of semantic objects

X the set of variables
K the set of input keys
T the set of all circuits
$U = \mathbb{N} \cup \mathbb{B} \cup () \cup U^*$ the set of core values
$V = \mathbb{N} \cup \mathbb{B} \cup () \cup (M\,V \times M\,T)^*$ the set of compound values

$M\,A = \mathbb{1} + A$
$C\,A = \mathbf{Env} \to \mathbf{In}^3 \to \mathbb{N}^3 \to \mathbb{1} + A \times \mathbf{Env} \times \mathbf{Out} \times \mathbb{N}^3$

$\mathbf{Env} = X \to M\,V \times M\,T$ value environments
$\mathbf{In} = K \to U$ inputs of local computation
$\mathbf{Out} = T^*$ streams of subcircuits

$[\![e]\!] : C\,(M\,V \times M\,T)$ the result of compilation of expression e

Types of semantic objects

X the set of variables
K the set of input keys
T the set of all circuits
$U = \mathbb{N} \cup \mathbb{B} \cup () \cup U^*$ the set of core values
$V = \mathbb{N} \cup \mathbb{B} \cup () \cup (M\,V \times M\,T)^*$ the set of compound values

$$M\,A = \mathbb{1} + A$$
$$C\,A = \mathbf{Env} \to \mathbf{In}^3 \to \mathbb{N}^3 \to \mathbb{1} + A \times \mathbf{Env} \times \mathbf{Out} \times \mathbb{N}^3$$

$\mathbf{Env} = X \to M\,V \times M\,T$ value environments
$\mathbf{In} = K \to U$ inputs of local computation
$\mathbf{Out} = T^*$ streams of subcircuits

$[\![e]\!] : C\,(M\,V \times M\,T)$ the result of compilation of expression e

Types of semantic objects

X the set of variables
K the set of input keys
T the set of all circuits
$U = \mathbb{N} \cup \mathbb{B} \cup () \cup U^*$ the set of core values
$V = \mathbb{N} \cup \mathbb{B} \cup () \cup (M\,V \times M\,T)^*$ the set of compound values

$$M\,A = \mathbb{1} + A$$
$$C\,A = \mathbf{Env} \to \mathbf{In}^3 \to \mathbb{N}^3 \to \mathbb{1} + A \times \mathbf{Env} \times \mathbf{Out} \times \mathbb{N}^3$$

$\mathbf{Env} = X \to M\,V \times M\,T$ value environments
$\mathbf{In} = K \to U$ inputs of local computation
$\mathbf{Out} = T^*$ streams of subcircuits

$[\![e]\!] : C\,(M\,V \times M\,T)$ the result of compilation of expression e

Examples of compilation rules

$$
\begin{aligned}
[\![\overline{n}]\!]\gamma\phi\nu &= \mathsf{pure}((\begin{cases}\mathsf{pure}\ n & \text{if } d = \textbf{@public}\\ \top & \text{otherwise}\end{cases}, \begin{cases}\mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } s = \textbf{\$post}\\ \top & \text{otherwise}\end{cases}), \gamma, \epsilon, \nu)\\
[\![x]\!]\gamma\phi\nu &= \mathsf{pure}(\gamma(x), \gamma, \epsilon, \nu)\\
[\![e_1 + e_2]\!]\gamma_0\phi\nu_0 &= \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0;\\
&\qquad ((\hat{v}_2, \hat{c}_2), \gamma_2, o_2, \nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1;\\
&\qquad \mathsf{pure}((\hat{s}, \hat{t}), \gamma_2, o_1 o_2, \nu_2)\\
&\quad \}\\
&\text{where } \hat{s} = \mathsf{do}\{i_1 \leftarrow \hat{v}_1;\ i_2 \leftarrow \hat{v}_2;\ \mathsf{pure}(i_1 + i_2)\},\\
&\qquad \hat{t} = \mathsf{do}\{c_1 \leftarrow \hat{c}_1;\ c_2 \leftarrow \hat{c}_2;\ \mathsf{pure}(\mathsf{node}[\mathsf{op}\,(+)](c_1, c_2))\}\\
[\![\textbf{assert}(e)]\!]\gamma_0\phi\nu_0 &= \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e]\!]\gamma_0\phi\nu_0;\ \mathsf{guard}(\hat{c}_1 \neq \top);\ \mathsf{pure}((\mathsf{pure}\,1, \top), \gamma_1, o_1 c_1, \nu_1)\}\\
&\text{where } \hat{c}_1 = \mathsf{pure}(c_1)
\end{aligned}
$$

Examples of compilation rules

$$
\begin{aligned}
[\![\overline{n}]\!]\gamma\phi\nu &= \mathsf{pure}((\begin{cases}\mathsf{pure}\ n & \text{if } d = \textbf{@public}\\ \top & \text{otherwise}\end{cases}, \begin{cases}\mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } s = \textbf{\$post}\\ \top & \text{otherwise}\end{cases}), \gamma, \epsilon, \nu)\\
[\![x]\!]\gamma\phi\nu &= \mathsf{pure}(\gamma(x), \gamma, \epsilon, \nu)\\
[\![e_1 + e_2]\!]\gamma_0\phi\nu_0 &= \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0;\\
&\qquad ((\hat{v}_2, \hat{c}_2), \gamma_2, o_2, \nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1;\\
&\qquad \mathsf{pure}((\hat{s}, \hat{t}), \gamma_2, o_1 o_2, \nu_2)\\
&\quad \}\\
&\text{where } \hat{s} = \mathsf{do}\{i_1 \leftarrow \hat{v}_1;\ i_2 \leftarrow \hat{v}_2;\ \mathsf{pure}(i_1 + i_2)\},\\
&\qquad \hat{t} = \mathsf{do}\{c_1 \leftarrow \hat{c}_1;\ c_2 \leftarrow \hat{c}_2;\ \mathsf{pure}(\mathsf{node}[\mathsf{op}\,(+)](c_1, c_2))\}\\
[\![\textbf{assert}(e)]\!]\gamma_0\phi\nu_0 &= \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e]\!]\gamma_0\phi\nu_0;\ \mathsf{guard}(\hat{c}_1 \neq \top);\ \mathsf{pure}((\mathsf{pure}\,1, \top), \gamma_1, o_1 c_1, \nu_1)\}\\
&\text{where } \hat{c}_1 = \mathsf{pure}(c_1)
\end{aligned}
$$

Examples of compilation rules

$$[\![\overline{n}]\!]\gamma\phi\nu = \mathsf{pure}((\begin{cases}\mathsf{pure}\ n & \text{if } d = \textbf{@public} \\ \top & \text{otherwise}\end{cases}, \begin{cases}\mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } s = \textbf{\$post} \\ \top & \text{otherwise}\end{cases}), \gamma, \epsilon, \nu)$$

$$[\![x]\!]\gamma\phi\nu = \mathsf{pure}(\gamma(x), \gamma, \epsilon, \nu)$$

$$\begin{aligned}[\![e_1\ +\ e_2]\!]\gamma_0\phi\nu_0 = \mathsf{do}\{&((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0; \\ &((\hat{v}_2, \hat{c}_2), \gamma_2, o_2, \nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1; \\ &\mathsf{pure}((\hat{s}, \hat{t}), \gamma_2, o_1 o_2, \nu_2) \\ \}& \\ \text{where } \hat{s} = \mathsf{do}\{&i_1 \leftarrow \hat{v}_1;\ i_2 \leftarrow \hat{v}_2;\ \mathsf{pure}(i_1 + i_2)\}, \\ \hat{t} = \mathsf{do}\{&c_1 \leftarrow \hat{c}_1;\ c_2 \leftarrow \hat{c}_2;\ \mathsf{pure}(\mathsf{node}[\mathsf{op}\,(+)](c_1, c_2))\}\end{aligned}$$

$$\begin{aligned}[\![\textbf{assert}(e)]\!]\gamma_0\phi\nu_0 = \mathsf{do}\{&((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e]\!]\gamma_0\phi\nu_0;\ \mathsf{guard}(\hat{c}_1 \neq \top);\ \mathsf{pure}((\mathsf{pure}\,1, \top), \gamma_1, o_1 c_1, \nu_1)\} \\ \text{where } \hat{c}_1 = \mathsf{pure}(c_1)&\end{aligned}$$

Examples of compilation rules

$$
\begin{aligned}
[\![\overline{n}]\!]\gamma\phi\nu &= \mathsf{pure}((\begin{cases}\mathsf{pure}\ n & \text{if } d = \textbf{@public} \\ \top & \text{otherwise}\end{cases}, \begin{cases}\mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } s = \textbf{\$post} \\ \top & \text{otherwise}\end{cases}), \gamma, \epsilon, \nu) \\
[\![x]\!]\gamma\phi\nu &= \mathsf{pure}(\gamma(x), \gamma, \epsilon, \nu) \\
[\![e_1 + e_2]\!]\gamma_0\phi\nu_0 &= \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0; \\
&\qquad ((\hat{v}_2, \hat{c}_2), \gamma_2, o_2, \nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1; \\
&\qquad \mathsf{pure}((\hat{s}, \hat{t}), \gamma_2, o_1 o_2, \nu_2) \\
&\quad \} \\
&\text{where } \hat{s} = \mathsf{do}\{i_1 \leftarrow \hat{v}_1;\ i_2 \leftarrow \hat{v}_2;\ \mathsf{pure}(i_1 + i_2)\}, \\
&\qquad \hat{t} = \mathsf{do}\{c_1 \leftarrow \hat{c}_1;\ c_2 \leftarrow \hat{c}_2;\ \mathsf{pure}(\mathsf{node}[\mathsf{op}\,(+)](c_1, c_2))\} \\
[\![\textbf{assert}(e)]\!]\gamma_0\phi\nu_0 &= \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e]\!]\gamma_0\phi\nu_0;\ \mathsf{guard}(\hat{c}_1 \neq \top);\ \mathsf{pure}((\mathsf{pure}\,1, \top), \gamma_1, o_1 c_1, \nu_1)\} \\
&\text{where } \hat{c}_1 = \mathsf{pure}(c_1)
\end{aligned}
$$

Examples of compilation rules

$$
\begin{array}{lll}
[\![\overline{n}]\!]\gamma\phi\nu & = & \mathsf{pure}((\begin{cases}\mathsf{pure}\ n & \text{if } d = \textbf{@public} \\ \top & \text{otherwise}\end{cases}, \begin{cases}\mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } s = \textbf{\$post} \\ \top & \text{otherwise}\end{cases}), \gamma, \epsilon, \nu) \\
[\![x]\!]\gamma\phi\nu & = & \mathsf{pure}(\gamma(x), \gamma, \epsilon, \nu) \\
[\![e_1 + e_2]\!]\gamma_0\phi\nu_0 & = & \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0; \\
 & & \quad ((\hat{v}_2, \hat{c}_2), \gamma_2, o_2, \nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1; \\
 & & \quad \mathsf{pure}((\hat{s}, \hat{t}), \gamma_2, o_1 o_2, \nu_2) \\
 & & \} \\
 & \text{where} & \hat{s} = \mathsf{do}\{i_1 \leftarrow \hat{v}_1;\ i_2 \leftarrow \hat{v}_2;\ \mathsf{pure}(i_1 + i_2)\}, \\
 & & \hat{t} = \mathsf{do}\{c_1 \leftarrow \hat{c}_1;\ c_2 \leftarrow \hat{c}_2;\ \mathsf{pure}(\mathsf{node}[\mathsf{op}\,(+)](c_1, c_2))\} \\
[\![\textbf{assert}(e)]\!]\gamma_0\phi\nu_0 & = & \mathsf{do}\{((\hat{v}_1, \hat{c}_1), \gamma_1, o_1, \nu_1) \leftarrow [\![e]\!]\gamma_0\phi\nu_0;\ \mathsf{guard}(\hat{c}_1 \neq \top);\ \mathsf{pure}((\mathsf{pure}\,1, \top), \gamma_1, o_1 c_1, \nu_1)\} \\
 & \text{where} & \hat{c}_1 = \mathsf{pure}(c_1)
\end{array}
$$

Examples of compilation rules (ctd.)

$$
\begin{aligned}
&[\![\textbf{if } e_1 \ \{\ e_2\ \} \textbf{ else } \{\ e_3\ \}]\!]\gamma_0\phi\nu_0 = \\
&\quad \mathsf{do}\{((\hat{v}_1,\hat{c}_1),\gamma_1,o_1,\nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0; \\
&\qquad \begin{cases} \mathsf{do}\{((\hat{v}_2,\hat{c}_2),\gamma_2,o_2,\nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1;\ \mathsf{pure}((\hat{v}_2,\hat{c}_2),\gamma_2,o_1o_2,\nu_2)\} & \text{if } \hat{v}_1 = \mathsf{pure}\,\mathbf{tt} \\ \mathsf{do}\{((\hat{v}_3,\hat{c}_3),\gamma_3,o_3,\nu_3) \leftarrow [\![e_3]\!]\gamma_1\phi\nu_1;\ \mathsf{pure}((\hat{v}_3,\hat{c}_3),\gamma_3,o_1o_3,\nu_3)\} & \text{if } \hat{v}_1 = \mathsf{pure}\,\mathbf{ff} \\ \mathsf{pure}((\top,\top),\gamma_1,o_1,\nu_1) & \text{otherwise} \end{cases} \\
&\quad \} \\
&[\![\textbf{wire } \{\ e\ \}]\!]\gamma\phi\nu = \\
&\quad \mathsf{do}\{((\hat{v}',\hat{c}'),\gamma',o',\nu') \leftarrow [\![e]\!]\gamma\phi\nu; \\
&\qquad \mathsf{pure}((\hat{v}', \begin{cases} \mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } \hat{v}' = \mathsf{pure}\,n,\ n \in \mathbb{N} \\ \mathsf{pure}(\mathsf{node}[\mathsf{con}(|b|)]) & \text{if } \hat{v}' = \mathsf{pure}\,b,\ b \in \mathbb{B} \\ \mathsf{pure}(\mathsf{node}[\mathsf{in}_d(\nu'_d)]) & \text{if } \hat{v}' = \top \end{cases}), \gamma', o', \nu'') \\
&\quad \} \\
&\quad \text{where } d \text{ is the domain of } e \text{ and } \nu'' = (d' \mapsto \begin{cases} \nu'_{d'} + 1 & \text{if } d' = d \\ \nu'_{d'} & \text{otherwise} \end{cases})
\end{aligned}
$$

Examples of compilation rules (ctd.)

$$
\begin{aligned}
&[\![\textbf{if } e_1 \ \{\ e_2\ \} \textbf{ else } \{\ e_3\ \}]\!]\gamma_0\phi\nu_0 = \\
&\quad \mathsf{do}\{((\hat{v}_1,\hat{c}_1),\gamma_1,o_1,\nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0; \\
&\qquad \begin{Bmatrix} \mathsf{do}\{((\hat{v}_2,\hat{c}_2),\gamma_2,o_2,\nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1;\ \mathsf{pure}((\hat{v}_2,\hat{c}_2),\gamma_2,o_1o_2,\nu_2)\} & \text{if } \hat{v}_1 = \mathsf{pure}\,\mathbf{tt} \\ \mathsf{do}\{((\hat{v}_3,\hat{c}_3),\gamma_3,o_3,\nu_3) \leftarrow [\![e_3]\!]\gamma_1\phi\nu_1;\ \mathsf{pure}((\hat{v}_3,\hat{c}_3),\gamma_3,o_1o_3,\nu_3)\} & \text{if } \hat{v}_1 = \mathsf{pure}\,\mathbf{ff} \\ \mathsf{pure}((\top,\top),\gamma_1,o_1,\nu_1) & \text{otherwise} \end{Bmatrix} \\
&\quad \}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{wire } \{\ e\ \}]\!]\gamma\phi\nu = \\
&\quad \mathsf{do}\{((\hat{v}',\hat{c}'),\gamma',o',\nu') \leftarrow [\![e]\!]\gamma\phi\nu; \\
&\qquad \mathsf{pure}((\hat{v}', \begin{Bmatrix} \mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } \hat{v}' = \mathsf{pure}\,n,\ n \in \mathbb{N} \\ \mathsf{pure}(\mathsf{node}[\mathsf{con}(|b|)]) & \text{if } \hat{v}' = \mathsf{pure}\,b,\ b \in \mathbb{B} \\ \mathsf{pure}(\mathsf{node}[\mathsf{in}_d(\nu'_d)]) & \text{if } \hat{v}' = \top \end{Bmatrix}, \gamma', o', \nu'') \\
&\quad \}
\end{aligned}
$$

where d is the domain of e and $\nu'' = (d' \mapsto \begin{Bmatrix} \nu'_{d'} + 1 & \text{if } d' = d \\ \nu'_{d'} & \text{otherwise} \end{Bmatrix})$

Examples of compilation rules (ctd.)

$$
\begin{array}{l}
[\![\textbf{if } e_1 \ \{\ e_2\ \} \textbf{ else } \{\ e_3\ \}]\!]\gamma_0\phi\nu_0 = \\
\quad \mathsf{do}\{((\hat{v}_1,\hat{c}_1),\gamma_1,o_1,\nu_1) \leftarrow [\![e_1]\!]\gamma_0\phi\nu_0; \\
\qquad \begin{Bmatrix} \mathsf{do}\{((\hat{v}_2,\hat{c}_2),\gamma_2,o_2,\nu_2) \leftarrow [\![e_2]\!]\gamma_1\phi\nu_1;\ \mathsf{pure}((\hat{v}_2,\hat{c}_2),\gamma_2,o_1o_2,\nu_2)\} & \text{if } \hat{v}_1 = \mathsf{pure}\,\mathbf{tt} \\ \mathsf{do}\{((\hat{v}_3,\hat{c}_3),\gamma_3,o_3,\nu_3) \leftarrow [\![e_3]\!]\gamma_1\phi\nu_1;\ \mathsf{pure}((\hat{v}_3,\hat{c}_3),\gamma_3,o_1o_3,\nu_3)\} & \text{if } \hat{v}_1 = \mathsf{pure}\,\mathbf{ff} \\ \mathsf{pure}((\top,\top),\gamma_1,o_1,\nu_1) & \text{otherwise} \end{Bmatrix} \\
\quad \} \\
[\![\textbf{wire } \{\ e\ \}]\!]\gamma\phi\nu \qquad\qquad = \\
\quad \mathsf{do}\{((\hat{v}',\hat{c}'),\gamma',o',\nu') \leftarrow [\![e]\!]\gamma\phi\nu; \\
\qquad \mathsf{pure}((\hat{v}', \begin{Bmatrix} \mathsf{pure}(\mathsf{node}[\mathsf{con}(n)]) & \text{if } \hat{v}' = \mathsf{pure}\,n,\ n \in \mathbb{N} \\ \mathsf{pure}(\mathsf{node}[\mathsf{con}(|b|)]) & \text{if } \hat{v}' = \mathsf{pure}\,b,\ b \in \mathbb{B} \\ \mathsf{pure}(\mathsf{node}[\mathsf{in}_d(\nu'_d)]) & \text{if } \hat{v}' = \top \end{Bmatrix}), \gamma', o', \nu'') \\
\quad \} \\
\quad \text{where } d \text{ is the domain of } e \text{ and } \nu'' = (d' \mapsto \begin{Bmatrix} \nu'_{d'} + 1 & \text{if } d' = d \\ \nu'_{d'} & \text{otherwise} \end{Bmatrix})
\end{array}
$$

Conclusion

- Zero-knowledge proofs conveniently specified in high-level programming languages and subsequently compiled into arithmetic circuits and/or transformed to other low-level representation that zero-knowledge proof techniques can use
- Strong static type system as a possible way to guarantee zero knowledge:
  - Separately distinguishing domains (privacy dimension) and stages (location of computation / reliability dimension)
  - Keeping track of effects
  - Domains, stages and effects not orthogonal but interrelated, making the type system complicated

Conclusion

- Zero-knowledge proofs conveniently specified in high-level programming languages and subsequently compiled into arithmetic circuits and/or transformed to other low-level representation that zero-knowledge proof techniques can use
- Strong static type system as a possible way to guarantee zero knowledge:
  - Separately distinguishing domains (privacy dimension) and stages (location of computation / reliability dimension)
  - Keeping track of effects
  - Domains, stages and effects not orthogonal but interrelated, making the type system complicated

Conclusion

- Zero-knowledge proofs conveniently specified in high-level programming languages and subsequently compiled into arithmetic circuits and/or transformed to other low-level representation that zero-knowledge proof techniques can use
- Strong static type system as a possible way to guarantee zero knowledge:
  - Separately distinguishing domains (privacy dimension) and stages (location of computation / reliability dimension)
  - Keeping track of effects
  - Domains, stages and effects not orthogonal but interrelated, making the type system complicated

Conclusion

- Zero-knowledge proofs conveniently specified in high-level programming languages and subsequently compiled into arithmetic circuits and/or transformed to other low-level representation that zero-knowledge proof techniques can use
- Strong static type system as a possible way to guarantee zero knowledge:
  - Separately distinguishing domains (privacy dimension) and stages (location of computation / reliability dimension)
  - Keeping track of effects
  - Domains, stages and effects not orthogonal but interrelated, making the type system complicated

Conclusion

- Zero-knowledge proofs conveniently specified in high-level programming languages and subsequently compiled into arithmetic circuits and/or transformed to other low-level representation that zero-knowledge proof techniques can use
- Strong static type system as a possible way to guarantee zero knowledge:
  - Separately distinguishing domains (privacy dimension) and stages (location of computation / reliability dimension)
  - Keeping track of effects
  - Domains, stages and effects not orthogonal but interrelated, making the type system complicated

Conclusion

- Zero-knowledge proofs conveniently specified in high-level programming languages and subsequently compiled into arithmetic circuits and/or transformed to other low-level representation that zero-knowledge proof techniques can use
- Strong static type system as a possible way to guarantee zero knowledge:
  - Separately distinguishing domains (privacy dimension) and stages (location of computation / reliability dimension)
  - Keeping track of effects
  - Domains, stages and effects not orthogonal but interrelated, making the type system complicated