zkInterface, a tool for zero-knowledge interoperability

Daniel Benarroch, Kobi Gurkan, Aurel Nicolas, Eran Tromer

February 6, 2019

[Add abstract *—Eran***]** ★

1 Overview

1.1 Background

[Rewrite: *—Eran***]** Implementing zero-knowledge proof constructions is not a trivial task and comes with diverse matters, as is extensively explained in the Implementation Track proceeding of the first ZKProof workshop. One of the requirements is to create a compiler of programs into constraint systems that are consumed by the proving system. ★

There are several trade-offs one can consider when designing general-purpose (front-end) compilers, leading to distinct frameworks, APIs, generality, etc. Today, existing compilers are implemented to work best with their corresponding (back-end) proving system (somemetimes more than one). For a comprehensive list of front-ends and back-ends, you can go to zkp.science.

These libraries are usually built end-to-end: they take in some program that defines the statement and generate or verify a proof, in many cases without explicitly exposing the constraint system compilation. Moreover, if the compilers can output intermediary files and configurations, they are usually native to the specific back-end. In practice this means that

- there is no portability between different proving systems and compilers, and
- it is not possible to compile a program using code from different frameworks

1.2 Goals

[Rewrite: *—Eran***]** ★

We aim to solve this issue, as seen in Figure 1, by creating a community standard proposal for the ZKProof effort around constraint system formatting, building upon the work done at the first ZKProof workshop.

We design and implement a standard rank-1 constraint system (R1CS) interface between front-ends and back-ends. Our design encompasses programmable instance and witness reductions, while capturing the parameters of the different components of the statement to be proven. Given that these statements can be large and difficult to build, developers usually build smaller components that are re-usable with different statements; these components are sometimes called "gadgets". With our zkInterface one can piece together programmatically the different components to form a complete statement.

Desiderata

- Interoperability across frameworks and programming languages
- The ability to write components that can be consumed by different frameworks

- Overhead of the R1CS construction and witness reduction should be linear compared to a native implementation of the same gadgets
- Design an extensible interface, for example to support non R1CS systems.

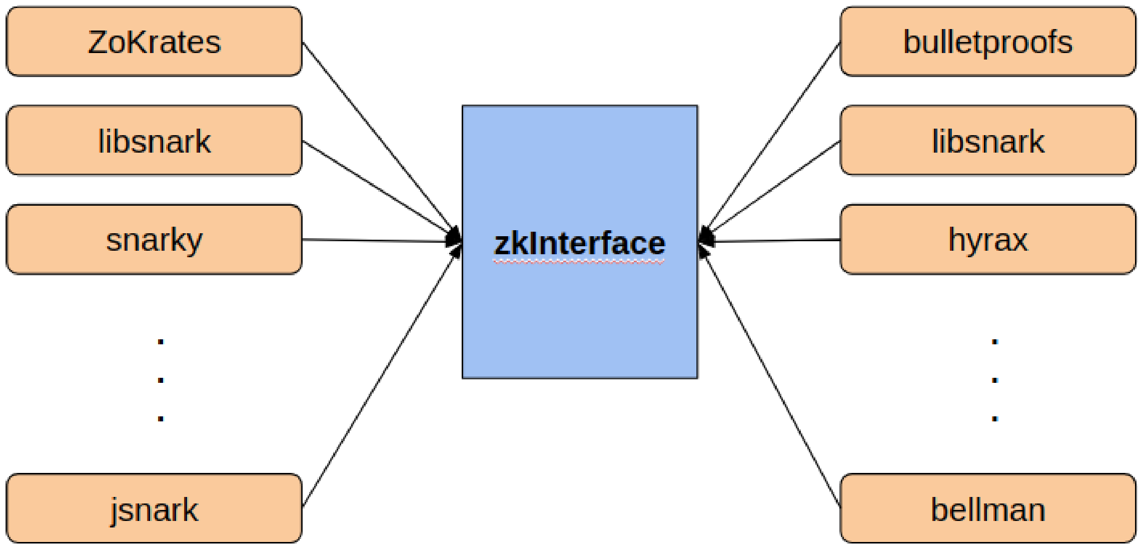

Figure 1: zkInterface **[Replace by diagram on slack —*Eran*]** ★

Scope and limitations. **[Explain what we won't do or defer to the future —*Eran*]** ★

We aim for the standard interface to be as generic as possible, including non-R1CS-based proving systems. However, the current proposal is more limited, mainly due to time constraints.

The standard that we propose can be seen in three different levels:

1. The first level defines
   - standard messages and their serialization that the caller and callee exchange,
   - an R1CS file format for the instance
   - a file format for the assignments.
2. The second level defines a simple C API that allows for the exchange of messages.
3. The third level defines the self-contained packaging of a component for its portable execution on different platforms.

This proposal is not aiming to standardize a language or framework for generating constraint systems, nor the way that components of the proving statement should be written. However, it is important to point that any such framework could use the proposed interface.

2 Design

2.1 Approach

zkInterface is a procedural, purely functional interface for zero-knowledge systems that enables cross-language interoperability via dynamic linking and shared memory. The current version, even if limiting, creates an interface based on R1CS formatting and offers the ability to abstractly craft a constraint system building from different components, possibly written in different frameworks, by determining how data should be written and read.

It can also be seen as a design tool for improved generation of constraints and usability, analogous to a portable binary format, since one can parametrize the functions calls and easily compose different functions, or components, that are not directly compatible.

It is important to point out that the interface can be called both to write a request or read a response by having an encoder at the front-end and a decoder at the back-end.

2.2 Architecture

[Describe the high-level design, introducing all the main concepts in a readable narrative and referring to them concretely by the identifier name in the sourcecode. —*Eran*] ★

Main functionality. The interface works across every zero-knowledge front-end and back-end, minimizing, when possible, the overhead of using a general format. This is achieved in several ways:

- By using a protoboard-like method for shared memory allocation, and thus preventing double-copying the data unnecessarily.
- By parametrizing the function calls to the different components so to take advantage of the specific context underlying those components.
- By using FlatBuffers, an efficient cross platform serialization library for different languages. This tool allows us to easily write ad-hoc parsers from scratch and has a very low overhead in shared memory, which can be used in regular function calls.

The two main purposes of the interface are the computations of the *instance reduction*, which generates a portable circuit or constraint system, and the *witness reduction*, which assigns values to the variables allocated in the instance reduction. We have designed the interface so that each of these two processes actually use the same exact routine, except with different message types.

Essentially, as seen in Figure 2, the caller of the interface can be both an application or a component that requires a sub-component, an abstraction that helps make the interface minimal. Say I want to compute a proof of set membership by using a Merkle Tree of hashes. Then, the flow is the following:

1. The application will call the Merkle Tree component that exists in some front-end framework, which starts allocating in memory the variables and constraints in the standard R1CS format.
2. For every hash computation needed to generate the path, the Merkle Tree will itself call a hasher sub-component, possibly from a different framework, by passing it the parameters, including the next free memory slot for allocating the hash constraints and variables.
3. The hash component will then allocate in memory the constraints and variables, to which the Merkle Tree component is oblivious (except the shared input / outputs: the input message and the output hash digest).
4. Specifically, for each call to the hash component, the input message is given as part of the request and the hash component sends the hash digest as part of the response. The rest of the variables are locally dealt with by the hash component but are shared in memory by all the components.

Note how the routine can be re-used by the witness reduction and deterministically assign the values to the respective variables in memory. Moreover, if needed, the constraint system can be outputed as a file containing a static rank-1 constraint system. One objection to using this routine design is that the component at the top level (i.e.: the Merkle Tree) cannot is waiting for the response of the sub-component (i.e.: the hasher component). This can have a cost in the efficiency of the circuit generation if we imagine a long enough chain of sub-calls that would cause a quadratic overhead. This is unlikely to happen in the current set of applications and circuits.

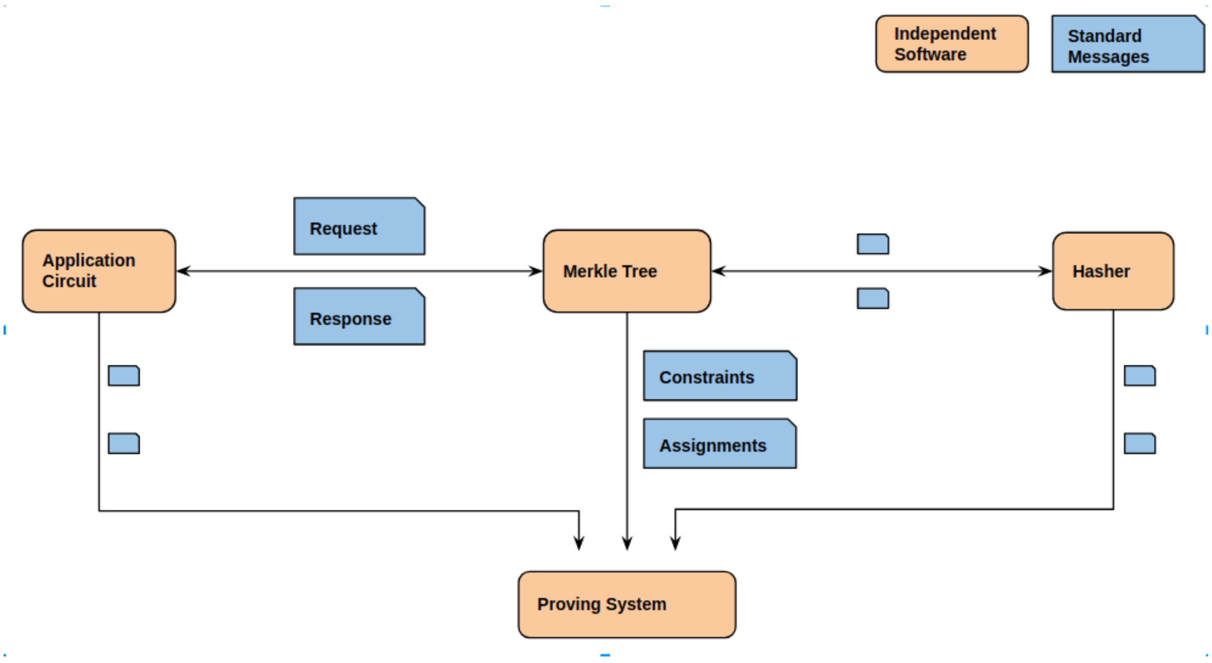

Figure 2: The flow of interaction between existing libraries and the interface

3 Specification

Interface Definition. The interface is defined as a FlatBuffers schema that describes the serialization of messages that the caller and callee can exchange.

The FlatBuffers system includes a simple interface definition language, support for all common programming languages, a clear evolution path for future extensions of the standard, and the possibility of very efficient implementations.

[Add ref: https://google.github.io/flatbuffers/ *—Aurel*] ★

The interface definition is provided in annex.

Request Call.

Callee Response.

C API.

Memory Allocation.

File Format.

3.1 Interface Definition

```
namespace Gadget;

union Message {
    R1CSConstraints,
    AssignedVariables,

    ComponentCall,
    ComponentReturn,
}

table Root {
    message :Message;
}

root_type Root;
file_identifier "zkp2";
file_extension "zkp2";


// == Types ==

// Variable ID = uint64
//
// Unique within a constraint system.
// Zero is a reserved special value.

// BigInt
//
// Big integers are represented as canonical little-endian byte arrays.
// Multiple big integers can be concatenated in a single array.
//
// Evolution plan:
// If a different representation of elements is to be supported in the future,
// it should use new fields, and omit the current canonical fields.
// This will allow past implementations to detect whether they are compatible.
//

/// Concrete variable values.
/// Used for linear combinations and assignments.
table VariableValues {
    /// The IDs of the variables being assigned to.
    variable_ids    :[uint64];

    /// Field Elements assigned to variables.
    /// Contiguous BigInts in the same order as variable_ids.
    ///
    /// The field in use is defined in `instance.field_order`.
    ///
    /// The size of an element representation is determined by:
    ///     element size = elements.length / variable_ids.length
    ///
    /// The element representation may be truncated and therefore shorter
    /// than the canonical representation. Truncated bytes are treated as zeros.
    elements        :[ubyte];
}

/// An R1CS constraint between variables.
table BilinearConstraint {
    // (A) * (B) = (C)
    linear_combination_a :VariableValues;
    linear_combination_b :VariableValues;
    linear_combination_c :VariableValues;
}

/// Description of a particular instance of a gadget.
table GadgetInstance {
    /// Which gadget to instantiate.
    /// Allows a library to provide multiple gadgets.
    gadget_name             :string;

    /// Incoming Variables to use as connections to the gadget.
    /// Allocated by the caller.
    /// Assigned by the caller in `Witness.incoming_elements`.
    incoming_variable_ids   :[uint64];

    /// Outgoing Variables to use as connections to the gadget.
    /// There may be no Outgoing Variables if the gadget is a pure assertion.
    /// Allocated by the caller.
    /// Assigned by the called gadget in `ComponentReturn.outgoing_elements`.
    outgoing_variable_ids   :[uint64];

    /// First free Variable ID before the call.
    /// The gadget can allocate new Variable IDs starting with this one.
    free_variable_id_before :uint64;

    /// The order of the field used by the current system.
    /// A BigInt.
    field_order             :[ubyte];

    /// Optional: Any static parameter that may influence the instance construction.
    /// Parameters can be standard, conventional, or specific to a gadget.
    /// Example: the depth of a Merkle tree.
    /// Counter-example: the Merkle path is not configuration (rather witness).
    configuration           :[CustomKeyValue];
}

/// Generic key-value for custom attributes.
table CustomKeyValue {
    key   :string;
    value :[ubyte];
}


// == Messages ==

/// Report constraints to be added to the constraints system.
/// To send to the stream of constraints.
table R1CSConstraints {
    constraints    :[BilinearConstraint];
}

/// Report local assignments computed by the gadget.
/// To send to the stream of assigned variables.
/// Does not include input and output variables.
table AssignedVariables {
    values :VariableValues;
}

table ComponentCall {
    /// All details necessary to construct the instance.
    /// The same instance must be provided for R1CS and assignment generation.
    instance            :GadgetInstance;

    /// Whether constraints should be generated.
    generate_r1cs       :bool;

    /// Whether an assignment should be generated.
    /// Provide witness values to the component.
    generate_assignment :bool;
    witness             :Witness;
}

/// Details necessary to compute an assignment.
table Witness {
    /// The values that the caller assigned to Incoming Variables.
    /// Contiguous BigInts in the same order as `instance.incoming_variable_ids`.
    incoming_elements :[ubyte];

    /// Optional: Any info that may be useful to the gadget to compute its assignments.
    /// Example: Merkle authentication path.
    info              :[CustomKeyValue];
}

/// Response after all R1CSConstraints or AssignedVariables have been sent.
table ComponentReturn {
    /// First variable ID free after the gadget call.
    /// A variable ID greater than all IDs allocated by the gadget.
    free_variable_id_after :uint64;

    /// Optional: Any info that may be useful to the caller.
    info                   :[CustomKeyValue];

    /// Optional: An error message. Null if no error.
    error                  :string;

    /// The values that the gadget assigned to outgoing variables, if any.
    /// Contiguous BigInts in the same order as `instance.outgoing_variable_ids`.
    outgoing_elements      :[ubyte];
}
```