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*]

[Copy relevant text from the Implementation Track, especially Advanced Interoperability *—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.

Scope and limitations. **[Rewrite: *—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

[Explain data flow; calling approach based on passing buffers; in-process or cross-process. Explain using FlatBuffers and why. Explain in general terms that we deal directly with variables in a global space, with lightweight coordination to allocate variables; and why (avoid duplication/rewriting etc. *—Eran*]

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.

[I'm not sure what this means: *—Eran*] 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.

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*]

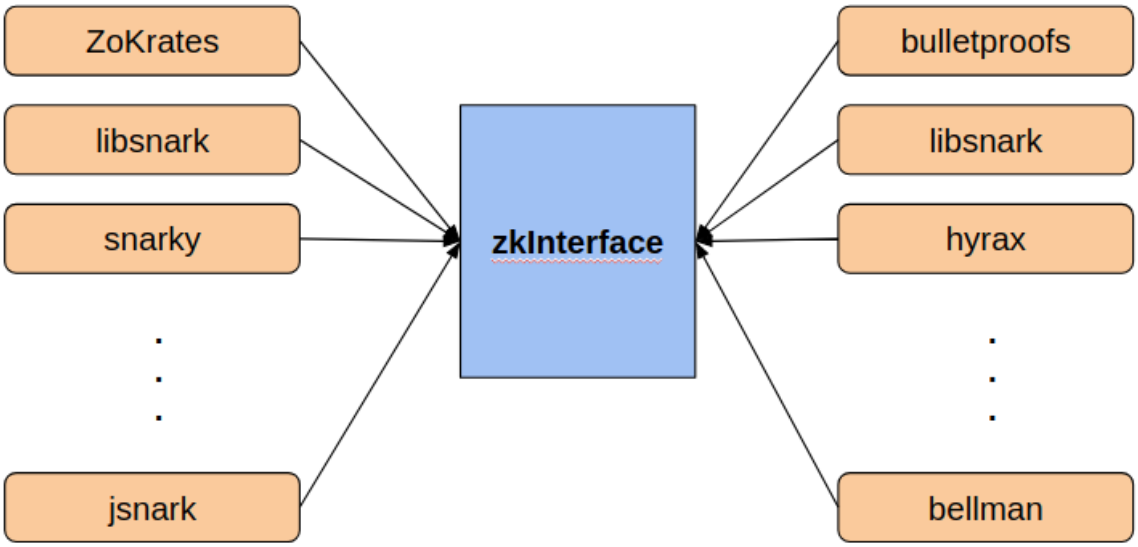

Figure 1: zkInterface **[Replace by diagram on slack** *—Eran*] ★

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.

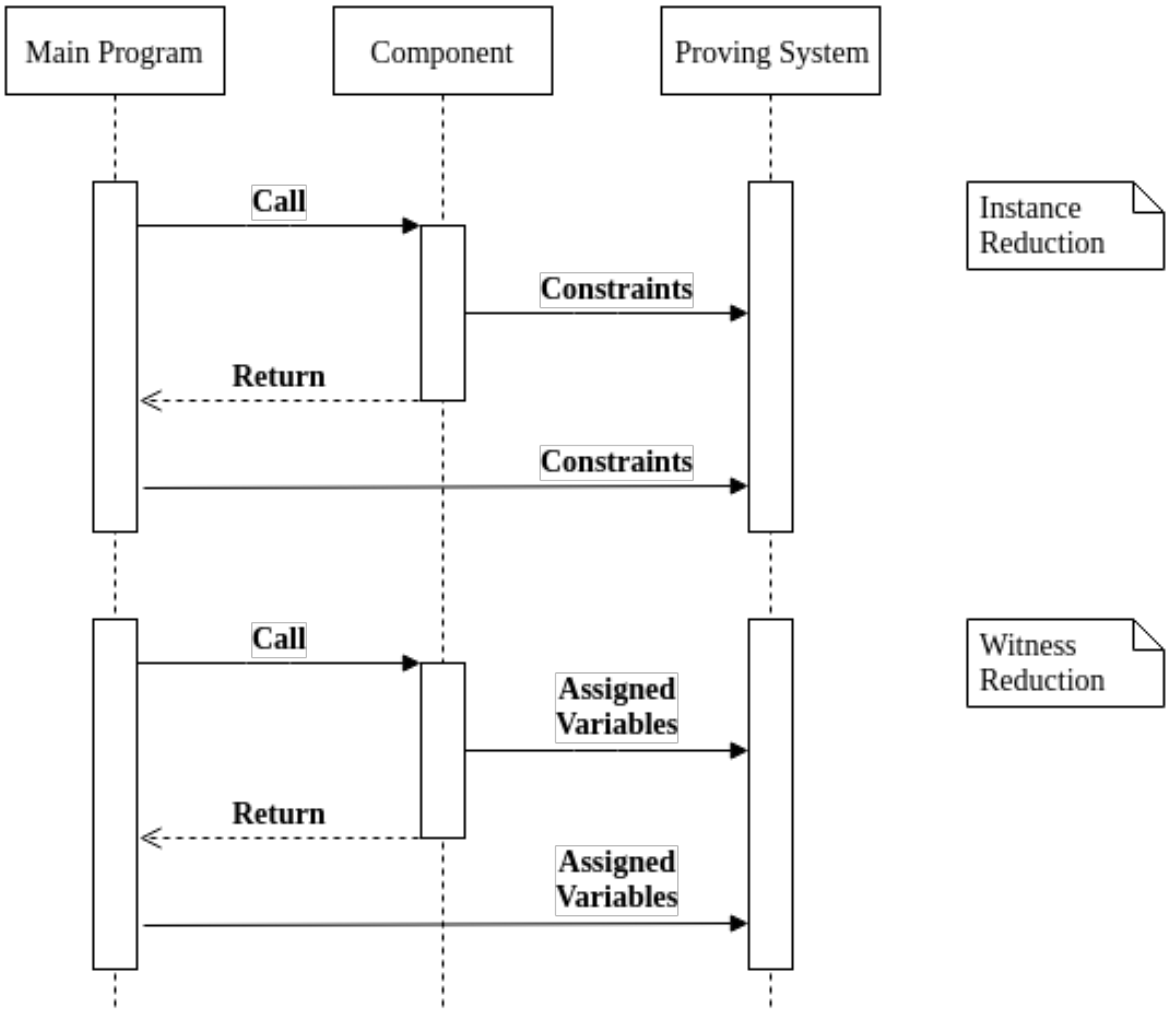

Figure 2: The flow of messages between libraries using the interface

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.

2.3 Calling convention and file format

[Explain how these are mostly implied by FlatBuffers. Give any additional information necessary beyond that. —*Eran*] ★

2.4 using zkInterface

[Explain, in concrete terms, what people need to do to use this standard. As a backend author? As a frontend framework author? As a guy who writes gadgets? As an integrator needing to bundle the dependencies? —*Eran*] ★

3 Specification

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

The interface is defined as a set of messages that the caller and callee can exchange. The definition is provided in annex. Refer to the inline documentation of each message and field.

Messages are described in a FlatBuffers schema.

Note The FlatBuffers system includes a simple interface definition language, a data layout specification, a clear evolution path for future extensions of the standard, support for all common programming languages, and the possibility of very efficient implementations. The specification of FlatBuffers can be found at https://google.github.io/flatbuffers/.

Messages Flow. The caller calls the component code with a single Call message. The component exits with a single Return message. This is a control flow analoguous to a function call in common programming languages.

The caller also provides a way for the component to send R1CSConstraints and AssignedVariables messages. This is an output channel distinct from the return message.

During instance reduction, a component may add any number of constraints to the constraint system by sending one or more R1CSConstraints messages.

During witness reduction, a component may assign values to variables by returning one or more AssignedVariables messages.

Note The design of constraints and assignments channels allows a component to call subcomponents itself. Messages from all (sub-)components can simply be sent separately without the need to aggregate them into a single message.

Moreover, an implementation can decouple the proving system from the logic of building constraints and assignments, by arranging for the constraints and assignments messages to be processed by the proving system, independently from the control logic.

Variables. All variables in a constraint system are assigned a numerical identifier unique within this system. Messages that contain constraints or assignments refer to variables by their unique ID.

Note This design allows implementations to aggregate and handle messages in a generic way, without any reference to the components or mechanisms that generated them.

[Must be consecutive or are gaps allowed? —*Aurel*] ★

Local Variables Allocation. A component may allocate a number of local variables to use in the internal implementation of the function that it computes. They are analoguous to stack variables in common programming languages.

The following protocol is used to allocate variable IDs that are unique within a whole constraint system.

- The caller must provide a numerical ID greater than all IDs that have already been allocated, called the Free-Before ID.
- The component may use the Free-Before ID and consecutive IDs as its local variables IDs.
- The component must return the next consecutive ID that it did not use, called the Free-After ID.
- The caller must treat IDs lesser than the Free-After ID as allocated by the component, and must not use them.

During instance reduction, the component can refer to its local variables in the constraints that it generates. The caller and other parts of the program must not refer to these variables.

During witness reduction, the component must assign values to its local variables.

Incoming/Outgoing Variables. The concept of incoming, outgoing variables arises when a program is decomposed into components. These variables serve as the functional interface between a component and its caller. They are analoguous to arguments and return values of functions in common programming languages. A variable is not inherently incoming, outgoing, nor local; rather, this is a convention in the context of a component call.

The caller provides the IDs of variables to be used as incoming and outgoing variables by the component.

During instance reduction, both the caller and the component can refer to these variables in the constraints that they generate. Other parts of the program may also refer to these same variables in their own contexts.

During witness reduction, the caller must assign values to incoming variables, and pass these values to the component. The component must compute values for outgoing variables, and return these values to the caller.

C Interface. A C interface to exchange messages is defined in annex. Refer to the inline documentation.

Stream and File Format. All messages are framed, meaning that they can be concatenated and distinguished in streams of bytes or in files. Messages must be prefixed by the size of the message not including the prefix, as a 4-bytes little-endian unsigned integer.

3.1 Interface Definition

Listing 1: gadget.fbs - Interface definition

```
namespace Gadget;

union Message {
    ComponentCall,
    ComponentReturn,

    R1CSConstraints,
```

```
    AssignedVariables,
}

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;
}

    /// 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 custom.
        /// Example: the depth of a Merkle tree.
        /// Counter-example: a Merkle path is not configuration (rather witness).
        configuration           :[KeyValue];
    }

    /// 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 assignments.
        /// Example: Merkle authentication path.
        info              :[KeyValue];
    }

    /// Generic key-value for custom attributes.
    table KeyValue {
        key   :string;
```

```
        value :[ubyte];
    }

/// 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                   :[KeyValue];

    /// 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];
}

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

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

/// 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;
}

    /// 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];
    }

    // type Variable ID = uint64
    //
    // IDs must be unique within a constraint system.
```

```
    // Zero is a reserved special value.

    // type 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.


table Root {
    message :Message;
}

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

Listing 2: gadget.h - C Interface

```
#ifndef GADGET_H
#define GADGET_H

#ifdef __cplusplus
extern "C" {
#endif


typedef bool (*gadget_callback_t)(
        void *context,
        unsigned char *response
);

bool gadget_request(
        unsigned char *request,

        gadget_callback_t result_stream_callback,
        void *result_stream_context,

        gadget_callback_t response_callback,
        void *response_context
);


#ifdef __cplusplus
} // extern "C"
#endif

#endif //GADGET_H
```