zkInterface, a tool for zero-knowledge interoperability

Daniel Benarroch, Aurel Nicolas, Eran Tromer

February 12, 2019

[Add abstract —*Eran*] **[Add bibliography and proper citations of the proceedings —*Daniel*]** ★ ★

1 Overview

In this work, as per the scope of the ZKProof effort **[cite: security / implementation track proceedings —*Daniel*]**, we propose a standard for interoperability for non-interactive proof systems (NIZKs) for general statements (NP) that use an R1CS/QAP-style constraint system representation. This includes many, though not all, of the practical general-purpose ZKP schemes currently deployed. While this focus allows us to define concrete formats for interoperability, we recognize that additional constraint system representation styles (e.g., arithmetic and Boolean circuits or algebraic constraints) are in use, and are within scope of future versions of the proposed standard. ★

There are many frontends for constructing constraint systems, and many backends which consume constraint systems (and variable assignments) to create or verify proofs. We focus on creating a message format that frontends and backends can use to communicate constraint systems and variable assignments. The design is aimed at simplicity, ease of implementation, compactness and avoiding hard-coded limits.

1.1 Background

Zero-Knowledge Proofs are cryptographic primitives that allow some entity (the prover) to prove to another party (the verifier) the validity of some statement or relation. Today there are many efficient constructions of NIZKs, each with different trade-offs, as well as several implementations of the proving systems. By standardizing zero-knowledge proofs, we aim to foster the proper use of the technology.

Every proving system can be divided **[cite: implementation track proceeding —*Daniel*]** into the backend, which is the portion of the software that contains the implementation of the underlying cryptographic protocol, and the frontend, which provides means to express statements in a convenient language, allowing to prove such statements in zero knowledge by compiling them into a low-level representation of the statement. ★

The backend of a proving system consists of the key generation, proving and verification algorithms. It proves statements where the instance and witness are expressed as variable assignments, and relations are expressed via low-level languages (such as arithmetic circuits, Boolean circuits, R1CS/QAP constraint systems or arithmetic constraint satisfaction problems). There are numerous such backends, including implementations of many of the schemes discussed in the Security Track proceeding **[cite: security track proceeding —*Daniel*]**. ★

The frontend consists of the following:

- The specification of a high-level language for expressing statements.

- A compiler that converts relations expressed in the high-level language into the low-level relations suitable for some backend(s). For example, this may produce an R1CS constraint system.
- Instance reduction: conversion of the instance in a high-level statement to a low-level instance (e.g., assignment to R1CS instance variables).
- Witness reduction: conversion of the witness to a high-level statement to a low-level witness (e.g., assignment to witness variables).
- Typically, a library of ”gadgets” consisting of useful and hand-optimized building blocks for statements.

Since the offerings and features of backends and frontends evolve rapidly, we refer the reader to the curated taxonomy at `https://zkp.science` for the latest information.

Currently, existing frontend are implemented to work best with their corresponding backend, the proving system is usually built end-to-end. The frontend compiles a statement into the native representation used by the cryptographic protocol in the backend, in many cases without explicitly exposing the constraint system compilation to the user. Moreover, if the compilers can output intermediary files and configurations, they are usually in a non-standard format. In practice this means that

- There is no portability between different backends and frontends, and
- It is not possible to generate a constraint system using different frontends

With this proposal we aim to solve this by creating a R1CS-based interface between frontends and backends. We add an explicit formatting layer between the frontends and backends that allows the user to “pick-and-chose” which existing frontend and backend they prefer. Furthermore, given the programatic design of our interface, a specific component, or gadget, can itself call a sub-component from a different frontend. This enables the use of more than one frontend to generate the complete statement.

1.2 Goals

[Rewrite: *—Eran*] **[use from proceeding “extensive interop”** *—Daniel*]

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

★
★
★

We design and implement a standard rank-1 constraint system (R1CS) interface between frontends and backends. Our design encompasses procedural instance and witness reductions, while capturing the parameters of the different components of the statement to be proven.

The following are stronger forms of interoperability which have been identified as desirable by practitioners, and are to be addressed by the ongoing standardization effort.

Statement and witness formats. In the R1CS File Format section and associated resources, we define a file format for R1CS constraint systems. There remains to finalize this specification, including instances and witnesses. This will enable users to have their choice of frameworks (frontends and backends) and streaming for storage and communication, and facilitate creation of benchmark test cases that could be executed by any backend accepting these formats.

Crucially, analogous formats are desired for constraint system languages other than R1CS.

Statement semantics, variable representation and mapping. Beyond the above, theres a need for different implementations to coordinate the semantics of the statement (instance) representation of constraint systems. For example, a high-level protocol may have an RSA signature as part of the statement, leaving ambiguity on how big integers modulo a constant are represented as a sequence of variables over a smaller field, and at what indices these variables are placed in the actual R1CS instance.

Precise specification of statement semantics, in terms of higher-level abstraction, is needed for interoperability of constraint systems that are invoked by several different implementations of the instance reduction (from high-level statement to the actual input required by the ZKP prover and verifier). One may go further and try to reuse the actual implementation of the instance reduction, taking a high-level and possibly domain-specific representation of values (e.g., big integers) and converting it into low-level variables. This raises questions of language and platform incompatibility, as well as proper modularization and packaging.

Note that correct statement semantics is crucial for security. Two implementations that use the same high-level protocol, same constraint system and compatible backends may still fail to correctly interoperate if their instance reductions are incompatible – both in completeness (proofs dont verify) or soundness (causing false but convincing proofs, implying a security vulnerability). Moreover, semantics are a requisite for verification and helpful for debugging.

Some backends can exploit uniformity or regularity in the constraint system (e.g., repeating patterns or algebraic structure), and could thus take advantage of formats and semantics that convey the requisite information.

At the typical complexity level of todays constraint systems, it is often acceptable to handle all of the above manually, by fresh re-implementation based on informal specifications and inspection of prior implementation. We expect this to become less tenable and more error prone as application complexity grows.

Witness reduction. Similar considerations arise for the witness reduction, converting a high-level witness representation (for a given statement) into the assignment to witness variables. For example, a high-level protocol may use Merkle trees of particular depth with a particular hash function, and a high-level instance may include a Merkle authentication path. The witness reduction would need to convert these into witness variables, that contain all of the Merkle authentication path data (encoded by some particular convention into field elements and assigned in some particular order) and moreover the numerous additional witness variables that occur in the constraints that evaluate the hash function, ensure consistency and Booleanity, etc.

The witness reduction is highly dependent on the particular implementation of the constraint system. Possible approaches to interoperability are, as above: formal specifications, code reuse and manual ad hoc compatibility.

Gadgets interoperability. At a finer grain than monolithic constraint systems and their assignments, there is need for sharing subcircuits and gadgets. For example, libsnark offers a rich library of highly optimized R1CS gadgets, which developers of several front-end compilers would like to reuse in the context of their own constraint-system construction framework.

While porting chunks of constraints across frameworks is relatively straightforward, there are challenges in coordinating the semantics of the externally-visible variables of the gadget, analogous to but more difficult than those mentioned above for full constraint systems: there is a need to coordinate or reuse the semantics of a gadgets externally-visible variables, as well as to coordinate or reuse the witness reduction function of imported gadgets in order to converts a witness into an assignment to the internal variables.

As for instance semantics, well-defined gadget semantics is crucial for soundness, completeness and verification, and is helpful for debugging.

Procedural interoperability. An attractive approach to the aforementioned needs for instance and witness reductions (both at the level of whole constraint systems and at the gadget level) is to enable one implementation to invoke the instance/witness reductions of another, even across frameworks and programming languages.

This requires communication not of mere data, but invocation of procedural code. Suggested approaches to this include linking against executable code (e.g., .so files or .dll), using some elegant and portable high-level language with its associated portable, or using a low-level portable executable format such as WebAssembly. All of these require suitable calling conventions (e.g., how are field elements represented?), usage guidelines and examples.

Beyond interoperability, some low-level building blocks (e.g., finite field and elliptic curve arithmetic) are needed by many or all implementations, and suitable libraries can be reused. To a large extent this is already happening, using the standard practices for code reuse using native libraries. Such reused libraries may offer a convenient common ground for consistent calling conventions as well.

1.3 Desiderata

1. Interoperability across frontend frameworks and programming languages.
2. Ability to write components that can be consumed by different frontends and backends.
3. Minimize copying and duplication of data.
4. The overhead of the R1CS construction and witness reduction should be low (and in particular, linear) compared to a native implementation of the same gadgets in existing frameworks.
5. Expose details of the backend's interface that are necessary for performance (e.g., constraint system representation and algebraic fields).
6. Aproach can be extended to support constraint systems beyond R1CS.

1.4 Scope, limitations and future work

[Rewrite: —*Eran*] **[need to discuss why R1CS: because it is a native language to many of the state-of-the-art constructions or are easily reducible to the native language. For those who are not native, they are complex and not generic to many constructions. —*Daniel*]** **[Need to pass the muthu test!! —*Daniel*]** ★ ★ ★

Rank-1 Constraint Systems are native to many of the state-of-the-art constructions or are reducible to the native low-level representation of the backend. Today these systems are in production and have a large user base

The Standard defined messages that the caller and callee exchange, including their serialization

Backend interoperability. Here, we do not aim to standardize the proof algorithms, the format proofs generated by a backend, or the format of the proving and verification keys – all of which would be required to achieve interoperability between backends. (See "Proof interoperability" and "Common reference strings" in **[ref the impl track]** . ◀

Programming language and frontend frameworks. We are intentionally agnostic about, and do not aim to standardize, the programming language and programming framework used by frontends.

The interface aims to be extensible with backwards compatibility, and we aim for future versions of the standard to be fully generic and to be as easy to use as possible. Possible extensions are the following:

Usability. **[what can we add about this? —*Daniel*]** ★

- A simple C API that allows for the exchange of messages would imply that one would not have to implement the standard message format in the specific programming language used by the frontend or backend.
- Self-contained packaging of a component would allow for portable execution of the components (or gadgets) on different platforms.
- going beyond R1CS (copy text from proceedings)

Generality. **[copy-paste from proceeding about semantics and about non-R1CS systems. —*Daniel*]** ★

- A message format that capture the specific semantics of the components.
- A statement representation that captures the specific structure of the statement, something that is not achieved with R1CS.
- **[Variable types, say for checking booleanity —*Daniel*]** ★

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,
   - a data format for R1CS constraints,
   - a data format for R1CS variables 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.

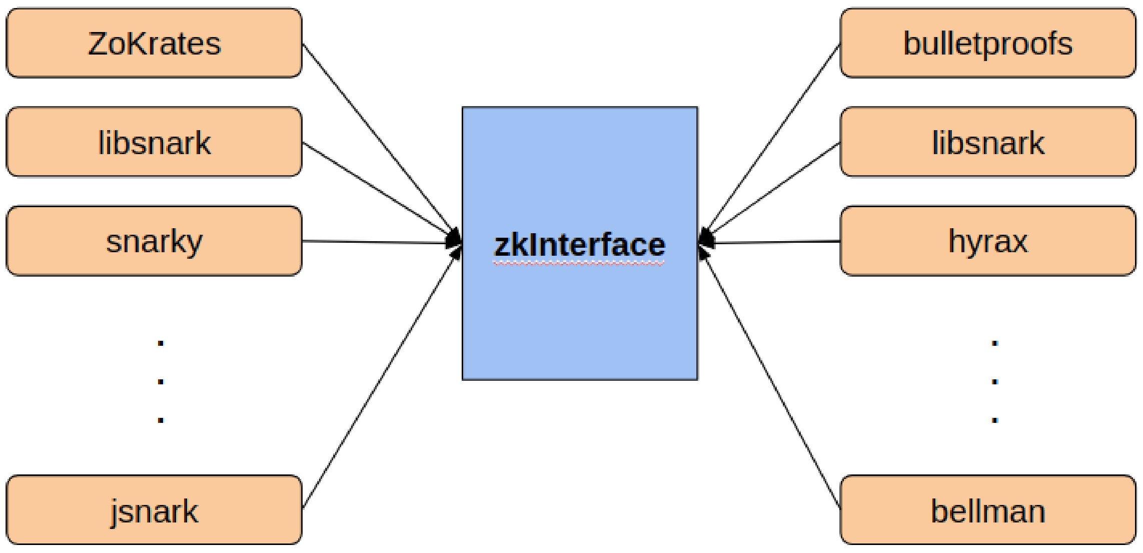

Figure 1: zkInterface **[Replace by diagram on slack —*Eran*]** ★

2 Design

2.1 Approach

zkInterface is a procedural, purely functional interface for zero-knowledge systems that enables cross-language interoperability. 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 defining how data should be written and read. It is independent of any particular proving systems.

The same interface can be used in two use-cases:

- To connect the construction and execution of a zero-knowledge program to a proving system. See Figure 2.
- To decompose a zero-knowledge program into multiple components that can be engineered separately. See Figure 3.

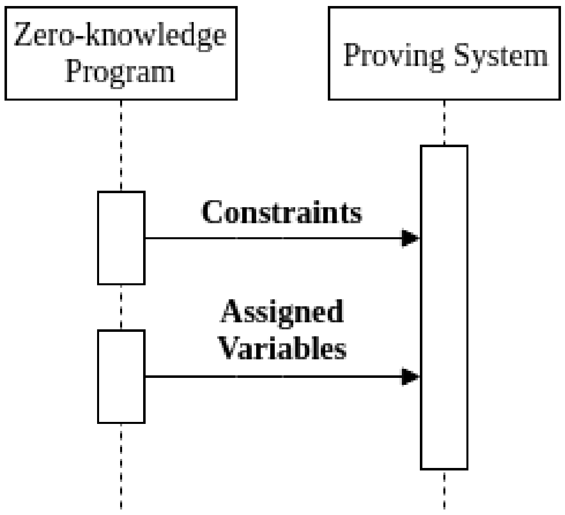

Figure 2: The interaction between a zero-knowledge program and a proving system.

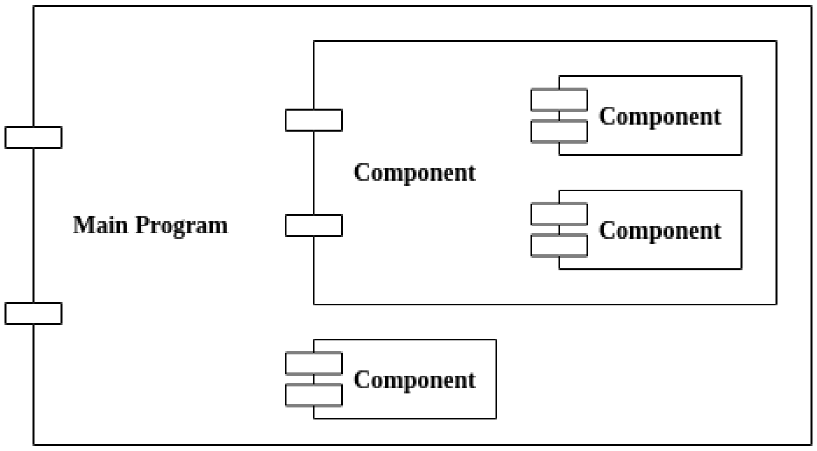

Figure 3: A zero-knowledge program built from multiple components.

Interface. The interaction between caller and components is based on exchanging messages. Messages are purely read-only data, which grants a great flexibility to implementations of components and applications.

Different parts of an application may be written in different programming languages, and interoperate through messages. These parts may be linked and executed in a single process, calling functions, and exchanging messages through buffers of shared memory. They may also run as separate processes, writing and reading messages in files or through pipes.

The set of messages is defined in Listing 1, using the FlatBuffers interface definition language. All messages and fields are defined in this schema.

Note The FlatBuffers system includes an interface definition language which implies a precise data layout at the byte level.

Code to write and read messages can be generated for all common programming languages. Code examples for C++ and for Rust are provided.

Multiple paths for evolution and extensions of the standard are possible, thanks to the flexibility and backward-compatibility features of the encoding.

The encoding is designed to require little to no data transformation, making it possible to implement the standard with minimal overhead in very large applications.

The specification of FlatBuffers can be found at https://google.github.io/flatbuffers/.

Instance and Witness Reductions. Instance reduction is the process of constructing a constraint system. Witness reduction is the process of assigning values to all variables in the system before generating a proof about concrete input values.

When using a proving system with pre-processing, instance reduction is performed once ahead of time and used in a trusted setup. In proving systems without pre-processing, instance reduction is used in proof verification. The standard supports both execution flows with similar messages.

2.2 Architecture

Messages Flow. The flow of messages is illustrated in Figure 4.

The caller calls the component code with a single ComponentCall message. The component exits with a single ComponentReturn 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 messages.

The caller can request an instance reduction, or a witness reduction, or both at once. This is controlled by the fields generate_r1cs and generate_assignment of ComponentCall messages.

During instance reduction, a component may add any number of constraints to the constraint system by sending one or more R1CSConstraints messages. The caller and other components may do so as well.

During witness reduction, a component may assign values to variables by sending 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. The constraint system reasons about variables, which are assigned a numerical identifier that are unique within a constraint system. The variable numbers are incrementally

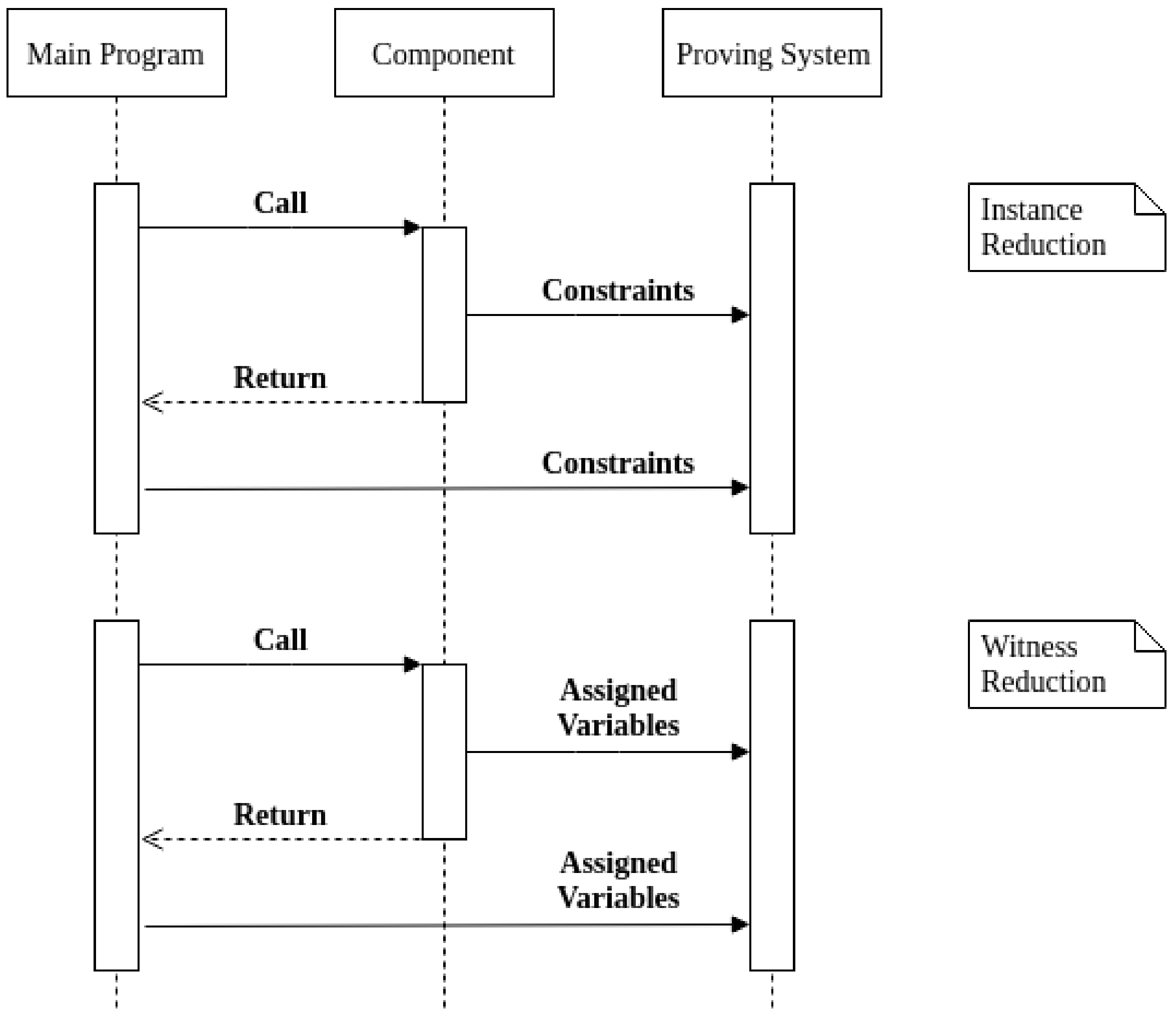

Figure 4: The flow of messages between libraries using the interface.

allocated in a global namespace. Messages that contain constraints or assignments refer to variables by this numeric ID. It is up to gadgets that create the constrain system to keep track of the semantics (and if desired, helpful symbolic names) of the variables that they deal with, and to allocate new variables for their internal use. The messages defined by the framework include a simple protocol for conveying the state of of the variable allocator (i.e., the first free variable number), and the identity of variables that tie a gadget to other gadgets.

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 R1CSConstraints messages that it generates. The caller and other parts of the program must not refer to these local variables.

During witness reduction, the component must assign values to its local variables by sending AssignedVariables messages.

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. There may be no outgoing variables if the component implements a pure assertion.

During instance reduction, both the caller and the component can refer to these variables in the R1CSConstraints messages 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 pass incoming values to the component in the ComponentCall message. The component must return outgoing values to the caller in the ComponentReturn message.

The caller is responsible for the assignment of values to both incoming and outgoing variables. How this is achieved depends on the caller and proving system, and on whether some variables are treated as public inputs of the zero-knowledge program.

In-memory execution - C API. The application may execute the code of a component in its own process.

The component exposes its functionnality as a function calleable using the C calling convention of the platform. The component code may be linked statically or be loaded from a shared library.

The application must prepare a ComponentCall message in memory, and implement one callback functions to receive the messages of the component, and call the component function with pointers to the message and the callbacks.

The component function reads the call message and performs its specific computation. It prepares the resulting messages (R1CSConstraints, AssignedVariables, or ComponentReturn) in memory, and calls the callbacks with pointers to these messages.

The function definition that implements this flow is defined in Listing 2 as a C header. Refer to the inline documentation for more details.

Multi-process execution - Stream and File Format. **[Explain what's already induced by FlatBuffers: explain that the framework, plus our .fbs file, specify how the aforementioned high-level messages are actually represented at the byte level, including message framing, message naming, etc.. —*Eran*]** ★ 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. **[Say something about streams - stdio/stdout, files, TCP connections, etc.? —*Eran*]** ★

2.3 Interface Definition

Listing 1: gadget.fbs - Interface definition

```
// This is a FlatBuffers schema.
// See https://google.github.io/flatbuffers/

namespace Gadget;

/// The messages that the caller and component can exchange.
union Message {
    ComponentCall,
    ComponentReturn,

    R1CSConstraints,
    AssignedVariables,
}

/// Caller calls a component.
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];
    }

/// Component returns to the caller. This is the final message
/// 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


/*  Callback functions.

    The caller implements these functions and passes function pointers to the component.
    The caller may also pass pointers to arbitrary opaque `context` objects of its choice.
    The component calls the callbacks with its response messages, and repeating the context pointe
 */
typedef bool (*gadget_callback_t)(
        void *context,
        unsigned char *response
);

/*  A function that implements a component.

    It receives a `ComponentCall` message, callbacks, and callback contexts.
    It calls `constraints_callback` one or more times with `constraints_context` and a `R1CSConstr
    It calls `assigned_variables_callback` one or more times with `assigned_variables_context` and
    Finally, it calls `return_callback` with `return_context` and a `ComponentReturn` message.
    The callbacks and the contexts pointers may be identical and may be NULL.

    The following memory management convention is used both for `call_gadget` and for the callbac
    All pointers passed as arguments to a function are only valid for the duration of this functi
    The code calling a function is responsible for managing the pointed objects after the functio
*/
bool call_gadget(
        unsigned char *call_msg,

        gadget_callback_t constraints_callback,
        void *constraints_context,

        gadget_callback_t assigned_variables_callback,
        void *assigned_variables_context,

        gadget_callback_t return_callback,
        void *return_context
);


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