zkInterface, a standard tool for zero-knowledge interoperability

Daniel Benarroch[1], Kobi Gurkan[1], Ron Kahat[1], Aurélien Nicolas[1], and Eran Tromer[2]

[1]QED-it
[2]QED-it, Columbia University, Tel Aviv University

June 10, 2019

Contents

1 Overview

This standard, part of the ZKProof Standardization effort [GKV+18, BGT+18, BCM+18], aims to facilitate interoperability between zero knowledge proof implementations, at the level of the low-constraint systems that are produced by frontends (and represent application-level statements) and consumed by cryptographic backends (which generate and verify the proofs). The high-level goal is to enable decoupling of frontends from backends, allowing application writers to choose the frontend most convenient for their functional and development needs and combine it with the backend that best matches their performance and security needs. This includes communicating constraint systems, communicating variable assignments (for production of proofs), and also construction of constraint systems out of smaller building blocks (gadgets) possibly implemented by different authors and frameworks.

This first revision focuses on non-interactive proof systems (NIZKs) for general statements (i.e., NP relations) represented in the R1CS/QAP-style constraint system representation[1]. 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 and algebraic constraints) are in use, and are within scope of future revisions.

An implementation of the zkInterface can be found in the following GitHub repository QED-it/zkinterface.

1.1 Background

Zero-Knowledge Proofs [GMR85] 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, as described in [BGT+18], can be divided 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. Some of these languages include arithmetic circuits [GKR08, WTS+18, GKM+18, MBKM], Boolean circuits [GMO16, CDG+17, AHIV17], R1CS/QAP constraint systems [GGPR13, PHGR13, BCG+13b, BCTV14, Gro16, BSCR+18, KZM+15, WZC+18, BBB+18], and arithmetic constraint satisfaction problems [BSCGT13, BSBHR18], among others. There are numerous such backends, including implementations of many of the schemes discussed in the Security Track proceeding [GKV+18].

The frontend consists of the following:

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

[1] See Appendix A for a definition of Rank 1 Constraint System

- 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 zkp.science for the latest information.

1.2 Motivation

Currently, existing frontends 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 an interface between frontends and backends, as seen in Figure 1. 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 programmatic design of our interface, a specific gadget, can itself call a sub-gadget from a different frontend. This enables the use of more than one frontend to generate the complete statement. At present this format is tailored for the R1CS statement representation, defined in Appendix A and discussed below.

1.3 Terminology

The terminology follows the Implementation Track proceeding [BGT+18], and any new terms and concepts will be defined accordingly. The R1CS statement representation is introduced and defined in Appendix A.

1.4 Goals

There are several forms of interoperability and we set some of these as goals for this standard. One such form is between different implementations of the same backend construction, providing an interoperable format for the proving and verification keys,

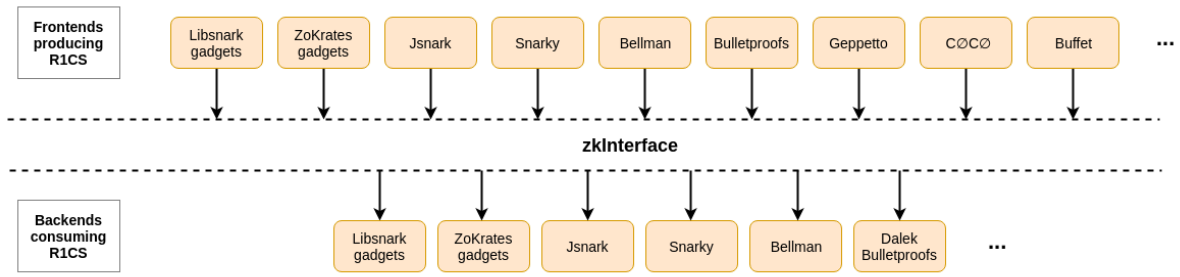

Figure 1: Interoperability between frontends and backends with zkInterface. For further reference, see zkp.science.

as well as the proof. Another form is between backends and frontends, which is the focus of this standard. The following are stronger forms of the latter kind of interoperability which have been identified as desirable by practitioners [BGT+18].

Statement instance and witness formats. Specifying a standard format for the statement instance and witness enables 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, there's 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 don't verify) or soundness (causing false but convincing proofs, implying a security vulnerability). Moreover, semantics are a requisite for verification and helpful for debugging. 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.

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.

Given the typical complexity level of today's 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. Our goal, however, is for the interface to handle the semantics of the gadgets, reducing the predisposition to error as application complexity grows. The following paragraphs expand on how the semantics should be considered for interoperability of gadgets.

Witness reduction. Similar considerations arise for the witness reduction, mapping a high-level witness representation for a given statement into the assignment to witness variables (as defined by the instance). 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. Moreover, it would also need to convert the numerous additional witness variables that occur in the constraints that evaluate the hash function, ensure consistency and Booleanity, amonh others.

Gadgets interoperability. Beyond using fixed, monolithic constraint systems and their assignments, there is a need for sharing subcircuits and gadgets. For example, libsnark offers a rich library of 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. Mainly, there is a need to coordinate or reuse the semantics of a gadget's externally-visible variables (those accessible by other gadgets), as well as the witness reduction function of imported gadgets in order to assign a witness into the internal variables of the gadget.

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.

1.5 Desiderata

The following are requirements that guided our design, within the large space of possibilities for achieving the above goals.

1. Interoperability across frontend frameworks and programming languages.

2. Ability to write gadgets 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. The approach can be extended to support constraint systems beyond R1CS.

1.6 Scope, limitations and possible extensions

Within the set of specifications to be standardized that enable the use of an interface between zero-knowledge proof systems, we have identified the minimal items needed to create a standard interface that meets the goals and desired requirements. The following list forms the scope of our proposed standard.

- Standard defined messages that the caller and callee exchange.
- The serialization of the messages.
- A protocol to build a constraint system from gadget composition.
- Technical recommendations for implementation.

Some limitations of the standard, with respect to interoperability, are the following.

Limitations. The following are not addressed by this standard:

Backend interoperability. We do not aim to standardize the proof algorithms, the format of the 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 [BGT+18].

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.

Extensions. The following are not covered by the present revision of the standard, but should be covered by future extensions. Thus, the standard should be flexible enough and easy to extend in a backward-compatible fashion, to achieve the following:

Other constraint system representations. Going beyond R1CS, we plan on supporting other constraint system representations that are native to some ZK backends. These include Boolean circuits, arithmetic circuits, and algebraic constraint systems.

Uniform constraint systems. Some backends can take advantage of uniformity in the constraint system (e.g., when some elements are repeated many times). Support the expression of such uniformity in the constraint system representation, so backends can utilize it.

Packaging. Describe self-contained packaging of a gadget would allow for portable execution of the gadgets on different platforms.

Typing. Enforcing properties of variables using a type system (e.g., a "boolean variable" type that ensures a variable is 0 or 1 even when working over a large field).

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 gadgets, 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 gadgets that can be engineered separately. See Figure 3.

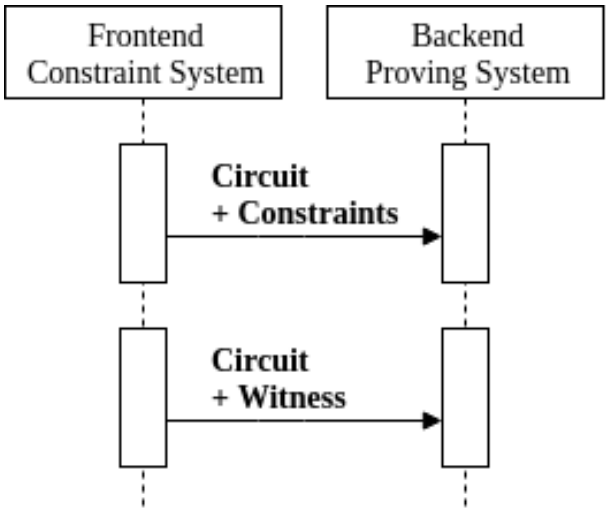

Figure 2: The interaction between a zero-knowledge program and a proving system.

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

Interoperability. 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 or sockets. Some implementation strategies are discussed in section 3.

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

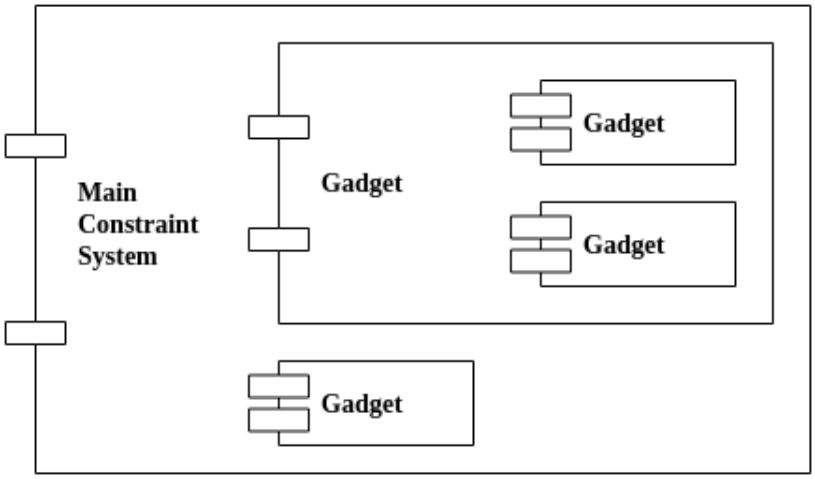

Figure 3: A zero-knowledge program built from multiple gadgets.

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. Examples are provided for C++ and for Rust.

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.

Messages must be prefixed by the size of the message not including the prefix, as a 4-bytes little-endian unsigned integer. This makes it possible to concatenate and distinguish messages in streams of bytes or in files.

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 the key generation. In proving systems without pre-processing, instance reduction is used in proof verification. The standard supports both execution flows.

2.2 Architecture

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

The caller calls the gadget code with a single `Circuit` message. This contains enough information for the gadget to generate its part of the constraint system or witness.

The gadget returns another `Circuit` message and exits. This contains information for the caller to continue building the rest of the constraint system or witness.

This is a control flow analogous to a function call in common programming languages.

The caller can request an instance reduction, or a witness reduction, or both at once. This is controlled by the fields `r1cs_generation` and `witness_generation` of

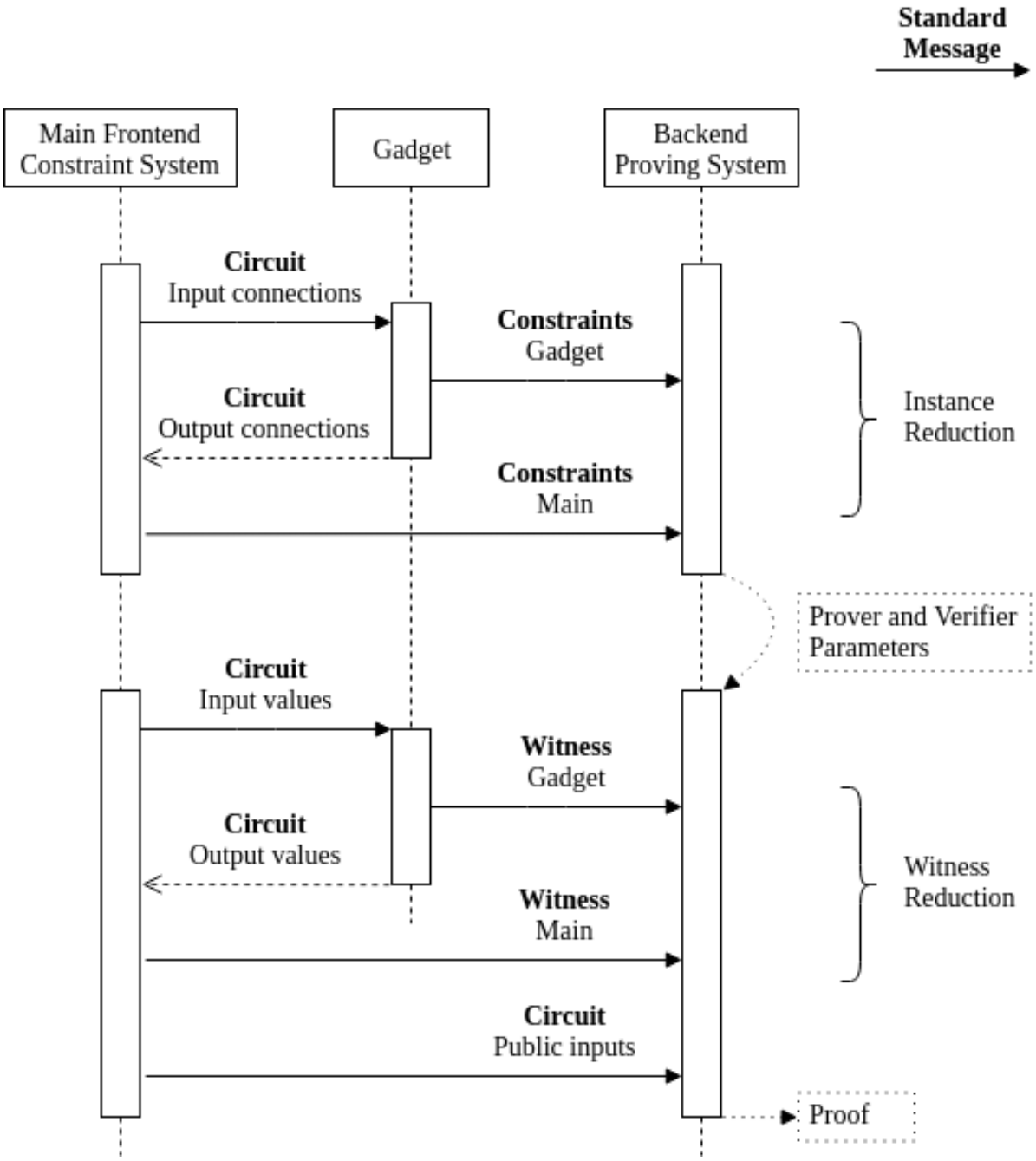

Figure 4: The flow of messages between libraries using the interface.

the caller's `Circuit` message.

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

During witness reduction, a gadget may assign values to variables by sending one or more `Witness` messages.

Messages Channels. The caller provides the gadget with the means to send messages, or output channels. This can be implemented in various ways depending on the application. The caller may arrange a distinct channel for each message type, and the

gadget must send messages to the appropriate channels.

Note. This design allows a gadget to call other gadgets itself. All `R1CSConstraints` or `Witness` messages from all (sub-)gadgets may be sent to a shared channel without the need to aggregate them into a single message, and independently of the call/return flow. 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 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 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 gadgets or mechanisms that generated them.

Local Variables Allocation. A gadget may allocate a number of local variables to use in the internal implementation of the function that it computes. They are analogous 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: the caller's `free_variable_id`.
- The gadget may use the `free_variable_id` and consecutive IDs as its local variables IDs.
- The gadget must return the next consecutive ID that it did not use: the returned `free_variable_id`.
- The caller must treat IDs lesser than the returned `free_variable_id` as allocated by the gadget, and must not use them.

During instance reduction, the gadget 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 gadget must assign values to its local variables by sending `Witness` messages.

Input/Output Connection Variables. The concept of input, output and local variables arises when a program is decomposed into gadgets. These variables serve as the functional interface between a gadget and its caller. They are analogous to arguments and return values of functions in common programming languages. A variable is not inherently input, output, nor local; rather, this is a convention in the context of a gadget call. There may be no output variables if the gadget implements a pure assertion over its inputs. When a `Circuit` message represents a complete circuit, the inputs are the public inputs to the proofs – also called the instance in the literature.

The following protocol is used to share input and output variables:

- Input and output variables are described in the `connections` structure of `Circuit` messages. This includes a required `variable_ids` array and an optional `values` array.
- The caller provides the `variable_ids` to be used as inputs by the gadget in its call message.
- The gadget returns the `variable_ids` to be used as outputs by the caller in its returned message.
- During witness reduction, the caller and gadget must also include the `values` to assign to input and output variables respectively.

During instance reduction, both the caller and the gadget 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.

The caller is responsible for communicating the values of both input and output variables to the backend. This is natural because the caller will typically use the output variables in its own logic. How this is achieved depends on the caller and proving system, and on whether some variables are treated as public inputs. The gadget should not include output values in `Witness` messages, as these would be redundant with its `Circuit.connections.values`.

2.3 Interface Definition

A copy of the messages definition is hosted at
https://github.com/QED-it/zkinterface/blob/master/zkinterface.fbs

Listing 1: Messages Definition

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

namespace zkinterface;


// ==== Message types that can be exchanged. ====

union Message {
    Circuit,
    R1CSConstraints,
    Witness,
}

/// A description of a circuit or sub-circuit.
/// This can be a complete circuit ready for proving,
/// or a part of a circuit being built.
table Circuit {

    /// Variables to use as connections to the sub-circuit.
    ///
    /// - Variables to use as input connections to the gadget.
    /// - Or variables to use as output connections from the gadget.
    /// - Variables are allocated by the sender of this message.
```

```
    /// - The same structure must be provided for R1CS and witness generations.
    /// - If `witness_generation=true`, variables must be assigned values.
    connections             :Variables;

    /// A variable ID greater than all IDs allocated by the sender of this message.
    /// The recipient of this message can allocate new IDs >= free_variable_id.
    free_variable_id        :uint64;

    // The fields below are required to call a backend or a gadget,
    // but optional in the response from a gadget.

    /// Whether a constraint system is being generated.
    /// Provide constraints in R1CSConstraints messages.
    r1cs_generation         :bool;

    /// Whether a witness is being generated.
    /// Provide the witness in `connections.values` and Witness messages.
    witness_generation      :bool;

    /// The largest element of the finite field used by the current system.
    /// A canonical little-endian representation of the field order minus one.
    /// See `Variables.values` below.
    field_maximum           :[ubyte];

    /// Optional: Any custom parameter that may influence the circuit construction.
    ///
    /// Example: function_name, if a gadget supports multiple function variants.
    /// Example: the depth of a Merkle tree.
    /// Counter-example: a Merkle path is not config and belongs in `connections.info`.
    configuration           :[KeyValue];
}

/// R1CSConstraints represents constraints to be added to the constraint system.
///
/// Multiple such messages are equivalent to the concatenation of `constraints` arrays.
table R1CSConstraints {
    constraints             :[BilinearConstraint];

    /// Optional: Any complementary info that may be useful.
    ///
    /// Example: human-readable descriptions.
    /// Example: custom hints to an optimizer or analyzer.
    info                    :[KeyValue];
}

/// Witness represents an assignment of values to variables.
///
/// - Does not include variables already given in `Circuit.connections`.
/// - Does not include the constant one variable.
/// - Multiple such messages are equivalent to the concatenation of `Variables` arrays.
table Witness {
    assigned_variables      :Variables;
}


// ==== Secondary Types ====

/// A single R1CS constraint between variables.
```

```
///
/// - Represents the linear combinations of variables A, B, C such that:
///       (A) * (B) = (C)
/// - A linear combination is given as a sequence of (variable ID, coefficient).
table BilinearConstraint {
    linear_combination_a    :Variables;
    linear_combination_b    :Variables;
    linear_combination_c    :Variables;
}

/// A description of multiple variables.
///
/// - Each variable is identified by a numerical ID.
/// - Each variable can be assigned a concrete value.
/// - In `Circuit.connections`, the IDs indicate which variables are
///   meant to be shared as inputs or outputs of a sub-circuit.
/// - During witness generation, the values form the assignment to the variables.
/// - In `BilinearConstraint` linear combinations, the values are the coefficients
///   applied to variables in a linear combination.
table Variables {

    /// The IDs of the variables.
    ///
    /// - IDs must be unique within a constraint system.
    /// - The ID 0 always represents the constant variable one.
    variable_ids            :[uint64];

    /// Optional: values assigned to variables.
    ///
    /// - Values are finite field elements as defined by `circuit.field_maximum`.
    /// - Elements are represented in canonical little-endian form.
    /// - Elements appear in the same order as variable_ids.
    /// - Multiple elements are concatenated in a single byte array.
    /// - The element representation may be truncated and its size shorter
    ///   than `circuit.field_maximum`. Truncated bytes are treated as zeros.
    /// - The size of an element representation is determined by:
    ///
    ///     element size = values.length / variable_ids.length
    values                  :[ubyte];

    /// Optional: Any complementary info that may be useful to the recipient.
    ///
    /// Example: human-readable names.
    /// Example: custom variable typing information (`is_bit`, ...).
    /// Example: a Merkle authentication path in some custom format.
    info                    :[KeyValue];
}

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


// ==== Flatbuffers details ====

// All message types are encapsulated in the FlatBuffers root table.
```

```
table Root {
    message                 :Message;
}
root_type Root;

// When storing messages to files, this extension and identifier should be used.
file_extension "zkif";
file_identifier "zkif"; // a.k.a. magic bytes.

// Message framing:
//
// All messages must be prefixed by the size of the message,
// not including the prefix, as a 4-bytes little-endian unsigned integer.
```

3 Implementation

The above section describes a protocol and the format of messages. Applications can execute, and exchange messages with gadgets and proving systems implementations in a variety of ways. We recommend different approaches to implementation in this section.

3.1 Execution

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

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

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

The gadget function reads the call message and performs its specific computation. It prepares the resulting messages of type `R1CSConstraints`, `Witness`, or `Circuit` in memory, and calls the callbacks with pointers to these messages.

A 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. Different parts of the application can be implemented as different programs, executed separately.

As specified in section 2.1, messages are framed and typed, and can be concatenated in a stream of bytes. It is therefore possible to connect multiple programs through UNIX-style pipes, or to arrange a program to write messages to a file for another program to read later. To process multiple messages from the same stream or file, a program reads the 4 bytes containing the size of the next message, allowing it to seek to the message after that.

A program that constructs a constraint system or implements a gadget should read a `Circuit` message from the standard input stream (stdin). It should write one or more messages of type `R1CSConstraints` or `Witness` to the standard output stream (stdout).

It should write a single `Circuit` message to the standard error stream (stderr, used as a control channel).

A program that implements a proving system should read messages of type `R1CSConstraints`, `Witness`, or `Circuit` from stdin, which should contain all the information needed to perform a pre-processing or to generate proofs.

3.2 Implementation Guide

Different tools can be implemented or adapted to use zkInterface. The following is a summary of their respective tasks. Refer to section 2.2 for more details.

Implementing a gadget.

- A gadget expects a `Circuit` message as input.
- If the field `r1cs_generation` is set, it outputs one or more `R1CSConstraints` messages.
- If the field `witness_generation` is set, it outputs one or more `Witness` messages.
- In both cases, it outputs a `Circuit` message.

Using a gadget in a frontend.

- To use a gadget, a frontend prepares a `Circuit` message for the gadget, and calls the gadget code. The frontend also decides how to handle replies from the gadget.
- When building a constraint system, the field `r1cs_generation` is set. The frontend will receive constraints in `R1CSConstraints` messages, and add them into the constraint system being built.
- When generating an assignment, the field `witness_generation` is set. The frontend will receive values to assign to the gadget variables in `Witness` messages, and include them into the complete assignment.
- In both cases, the frontend will receive a `Circuit` message from the gadget, which it can use in the rest of the process.

Implementing a frontend for interoperability with standard backends.

- A frontend describes a constraint system in a `Circuit` message. The incoming variables are interpreted as public inputs. There are no outgoing variables in this case.
- When building a constraint system, the frontend outputs all constraints in one or more `R1CSConstraints` messages.
- When generating a proof, the frontend outputs a complete assignment in one or more `Witness` messages.

Implementing a backend for interoperability with standard frontends.

- A backend initializes a constraint system using a `Circuit` message. The incoming variables are interpreted as public inputs. There are no outgoing variables in this case.
- The backend loads all constraints from `R1CSConstraints` messages. If the backend implements a pre-processing SNARK scheme, this is sufficient to perform a setup.
- To generate a proof, the backend loads a complete assignment from `Witness` messages.

Demonstration. Multiple example implementations are provided. This code is distributed under the MIT license at https://github.com/QED-it/zkinterface. More links will be found on this page.

A number of C++ helper functions are used to interoperate with libsnark protoboard objects. A SHA256 gadget from libsnark/gadgetlib1 is encapsulated with the message-based interface of section 2.

A set of Rust helper functions are available to write and read messages. A test program written in Rust demonstrates in-memory execution using the method in section 3.1, and includes helper functions to process messages.

A demonstration backend and frontend compiled independently to WebAssembly and running in web browsers is hosted at https://github.com/QED-it/zkinterface-wasm.

Listing 2: zkinterface.h - C Interface

```
#ifndef ZKINTERFACE_H
#define ZKINTERFACE_H
#ifdef __cplusplus
extern "C" {
#endif


/*  Callback functions.

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

/*  A function that implements a gadget.

    It receives a `Circuit` message, callbacks, and callback contexts.
    It calls `constraints_callback` zero or more times with
    `constraints_context` and a `R1CSConstraints` message.
    It calls `witness_callback` zero or more times with
    `witness_context` and a `Witness` message.
    Finally, it calls `return_callback` once with `return_context` and a
```

```
    return `Circuit` 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 callbacks. All pointers passed as arguments to a function are
    only valid for the duration of this function call. The caller of a function
    is responsible for managing the pointed objects after the function returns.
*/
bool call_gadget(
        unsigned char *call_msg,

        gadget_callback_t constraints_callback,
        void *constraints_context,

        gadget_callback_t witness_callback,
        void *witness_context,

        gadget_callback_t return_callback,
        void *return_context
);


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

References

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2087–2104. ACM, 2017.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[BCG+13a] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, pages 90–108, 2013. Extended version: [BCG+13b].

[BCG+13b] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge (extended version). Cryptology ePrint Archive, Report 2013/507, 2013. URL: `http://eprint.iacr.org/2013/507`.

[BCM+18] Daniel Benarroch, Ran Canetti, Andrew Miller, Shashank Agrawal, Tony Arcieri, Vipin Bharathan, Josh Cincinnati, Joshua Daniel, Anuj Das Gupta, Angelo De Caro, Michael Dixon, Maria Dubovitskaya, Nathan George,

Brett Hemenway Falk, Hugo Krawczyk, Jason Law, Anna Lysyanskaya, Zaki Manian, Eduardo Morais, Neha Narula, Gavin Pacini, Jonathan Rouach, Kartheek Solipuram, Mayank Varia, Douglas Wikstrom, and Aviv Zohar. Applications track proceeding. Technical report, ZKProof Standards, Berkeley, CA, May 2018. https://zkproof.org/documents.html.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security 2014*, pages 781–796, 2014. URL: http://eprint.iacr.org/2013/879.

[BGT+18] Sean Bowe, Kobi Gurkan, Eran Tromer, Benedikt Bünz, Konstantinos Chalkias, Daniel Genkin, Jack Grigg, Daira Hopwood, Jason Law, Andrew Poelstra, abhi shelat, Muthu Venkitasubramaniam, Madars Virza, Riad S. Wahby, and Pieter Wuille. Implementation track proceeding. Technical report, ZKProof Standards, Berkeley, CA, May 2018. https://zkproof.org/documents.html.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 401–414. ACM, 2013.

[BSCR+18] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent succinct arguments for r1cs. *IACR ePrint*, 828, 2018.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1825–1842. ACM, 2017.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT '13*, pages 626–645, 2013. URL: https://eprint.iacr.org/2012/215.

[GKM+18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. Cryptology ePrint Archive, Report 2018/280, 2018. https://eprint.iacr.org/2018/280.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for Muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC '08, pages 113–122, 2008.

[GKV+18] Jens Groth, Yael Kalai, Muthu Venkitasubramaniam, Nir Bitansky, Ran Canetti, Henry Corrigan-Gibbs, Shafi Goldwasser, Charanjit Jutla, Yuval Ishai, Rafail Ostrovsky, Omer Paneth, Tal Rabin, Mariana Raykova, Ron Rothblum, Alessandra Scafuro, Eran Tromer, and Douglas Wikström. Security track proceeding. Technical report, ZKProof Standards, Berkeley, CA, May 2018. https://zkproof.org/documents.html.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 1069–1083, 2016.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *STOC 1985*, pages 291–304, 1985. URL: https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Proof%20Systems/The_Knowledge_Complexity_Of_Interactive_Proof_Systems.pdf.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *proc. Eurocrypt '16, Part II*, pages 305–326, 2016.

[KZM+15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, H Chan, C PAPAMAN-THOU, R Pass, SHELAT ABHI, and EC SHI. C0c0: A framework for building composable zero-knowledge proofs. Technical report, Cryptology ePrint Archive, Report 2015/1093, 2015. http://eprint.iacr. org . . . , 2015.

[MBKM] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013. URL: https://eprint.iacr.org/2013/279.

[WTS+18] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943. IEEE, 2018.

[WZC+18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 675–692, 2018.

A R1CS Definition

Background: statement representations. Many known zk-SNARKs are generic and can express any *NP statement* (i.e., any statement expressed as "for the instance x there exists a witness r such that relation D holds on x and w" where the relation D is efficiently computable) [GKV+18]. In principle this already includes convenient high-level representations, and indeed there are research prototypes, for example, for SNARKs supporting general C programs [BCG+13a]. However, each zk-SNARK cryptographic

construction has a "native language" which is a specific type of NP languages. Anything else needs to be converted (reduced) to the native representation, with some overhead. Here we discuss "R1CS", one such low-level native representation, and how it can be used to efficiently express useful NP statements to be proven by zk-SNARK constructions.

Moreover, expressing NP statements for SNARKs differs from usual programming (whether in JavaScript or assembly) in the following sense. Usual programming aims to *compute* the output of some function of the given inputs, whereas the NP statement's relation is already given the instance (e.g., the claimed output of a function on some input), and aim to *check* its correctness given some additional information from the witness. This affects the programming style and creates opportunities for important optimizations, such as providing additional advice within the witness that aid the checking.

Constraint systems. Today's best zk-SNARK constructions have a native language consisting of *constraint systems*, i.e., a set of constraints on the instance and witness variables.[2] The most common such language is *Rank 1 Constraint System (R1CS)*, also known as *Bilinear Constraint Systems (BCS)*, where each constraint is a (bilinear) quadratic equation over a large finite field. This is the native language of many efficient zk-SNARKs, such as [GGPR13, PHGR13, BCG+13a, BCTV14, Gro16] and many others (see [BGT+18]), and is used in practical deployments. This style of constraint system was introduced in [GGPR13].[3] The variant defined below is very similar to the definition in [BCG+13b, Appendix E], used in libsnark.[4]

An R1CS reasons about two vectors: the *instance* consisting of n_{x} elements and denoted $\vec{x} = (x_1, \ldots, x_{n_{\mathsf{x}}})$, and the *witness* consisting of n_{w} elements and denoted $\vec{w} = (w_1, \ldots, w_{n_{\mathsf{w}}})$. The constraint system says that the two are related by some number of constraints, n_{c}, each of which is a quadratic equation of a specific form. All of the elements and operations are over a large prime field $\mathbb{F}_q$, which we will represent here as the integers modulo a large prime p.[5]

Definition. Let $n_{\mathsf{x}}, n_{\mathsf{w}}, n_{\mathsf{c}}$ be positive integers, and $n_{\mathsf{z}} = n_{\mathsf{x}} + n_{\mathsf{w}} + 1$. A ***Rank 1 Constraint System (R1CS)***, over $\mathbb{F}_q$ is a tuple $\mathcal{S} = \left((\vec{a}_j, \vec{b}_j, \vec{c}_j)_{j=1}^{n_{\mathsf{c}}}, n_{\mathsf{x}}\right)$ where $\vec{a}_j, \vec{b}_j, \vec{c}_j \in \mathbb{F}_q^{n_{\mathsf{z}}}$. Such a system $\mathcal{S}$ is ***satisfiable*** with an input $\vec{x} \in \mathbb{F}_q^{n_{\mathsf{x}}}$ if the following is true:

For these $\mathcal{S} = \left((\vec{a}_j, \vec{b}_j, \vec{c}_j)_{j=1}^{n_{\mathsf{c}}}, n_{\mathsf{x}}\right)$ and $\vec{x} \in \mathbb{F}_q^{n_{\mathsf{x}}}$;
there exists a witness $\vec{w} \in \mathbb{F}_q^{n_{\mathsf{w}}}$;
such that $\left\langle \vec{a}_j, \vec{z} \right\rangle \cdot \left\langle \vec{b}_j, \vec{z} \right\rangle = \left\langle \vec{c}_j, \vec{z} \right\rangle$ for all for $j \in [n_{\mathsf{c}}]$, where $\vec{z} = (1, \vec{x}, \vec{w}) \in \mathbb{F}_q^{n_{\mathsf{z}}}$.

Above, $\langle \cdots, \cdots \rangle$ denotes inner product of vectors over $\mathbb{F}_q$, and $\cdot$ denotes multiplication in $\mathbb{F}_q$.

[2] Colloquially, zk-SNARK developers often refer to constraint systems as "circuits", but as we shall soon see, the typical constraint system formalism is more general than Boolean or arithmetic circuits.

[3] The initial works [GGPR13, PHGR13] used an equivalent formulation called *Quadratic Arithmetic Programs (QAP)*, where the constraints are expressed in terms of polynomials. Later works tend to abstract this away, and do the conversion into polynomials internally.

[4] To be precise, "R1CS", as defined here, is the same as the "system of rank-1 quadratic equations" defined in [BCG+13b, Appendix E], with a minor change in notation. In defining BCS we let $\vec{w}$ denote just the witness, whereas [BCG+13b] let $\mathbf{w}$ denote the concatenation of the instance and the witness (and thus added corresponding consistency checks between $\mathbf{w}$ and the instance $\mathbf{x}$). These are also essentially identical to the "Rank 1 Constraint System (R1CS)" of libsnark.

[5] The specific prime, and the representation of field elements, are related to the elliptic curve used in the QAP-based zkSNARK constructions.

For example, the R1CS

$$\mathcal{S} = \left(\begin{pmatrix} \vec{a}_1 = (0,3,0,0),\ \vec{b}_1 = (1,1,0,0),\ \vec{c}_1 = (4,0,5,0) \\ \vec{a}_2 = (0,1,2,0),\ \vec{b}_2 = (6,0,1,0),\ \vec{c}_2 = (7,0,0,1) \end{pmatrix}, 1 \right) \tag{1}$$

represents the two bilinear constraints

$$\begin{aligned} \left\langle (0,3,0,0), \vec{z} \right\rangle \cdot \left\langle (1,1,0,0), \vec{z} \right\rangle &= \left\langle (4,0,5,0), \vec{z} \right\rangle \\ \left\langle (0,1,2,0), \vec{z} \right\rangle \cdot \left\langle (6,0,1,0), \vec{z} \right\rangle &= \left\langle (7,0,0,1), \vec{z} \right\rangle \end{aligned} \quad \text{where } \vec{z} = (1, x_1, w_1, w_2) \tag{2}$$

or simply:

$$\begin{aligned} 3x_1 \cdot (1 + x_1) &= 5w_1 + 4 \\ (x_1 + 2w_1) \cdot (w_1 + 6) &= w_2 + 7 \end{aligned} \tag{3}$$

By definition, this system $\mathcal{S}$ is satisfiable with input x_1 if the following is true:

For the instance x_1;
there exist a witness w_1, w_2;
such that

$$\begin{aligned} 3x_1 \cdot (1 + x_1) &= 5w_1 + 4 \\ (x_1 + 2w_1) \cdot (w_1 + 6) &= w_2 + 7 \end{aligned}$$

When we prove this NP statement using zk-SNARK, the instance will be the public input that is visible to the verifer, and the witness will be the additional data that is known only to the prover, and used in the construction of the proof, but whose privacy is protected by the zero-knowledge property.

Complexity measure. For backends that consume R1CS natively, the cost of producing the zk-SNARK proofs (in time and memory) typically depends mostly on the **number of constraints**, n_c.

In particular, multiplying variables (or linear combinations thereof) is "expensive" (each such multiplication needs a new constraint and thus increases n_c), but linear combinations (i.e., additions and multiplications by constants) come "for free" in each constraint.