The Verifiable Computing System

The system described below supports verifiable computing paradigm.

Terminology

1. **Program [P]:** This is a arithmetic function that takes N inputs and produces M outputs.
2. **HLL Program [HLL(P)]:** This is a high level representation of a Program P. For eg, as ANSI C or SFDL Code
3. **Circuit [CIRC(P)]:** This is a intermediate representation of a program P in the form of an arithmetic circuit consisting of input/output/intermediate wires and gates (addition, multiplication etc.)
4. **QAP Constraints [QAP(P)]**: This is a low-level representation of a program P in the form of quadratic constraints.
5. **R1CS Constraints [R1CS(P)]:** This is a low-level representation of a program P in the form of vectors.
6. **Input (x):** This is a vector of values that is to be evaluated by the Program P.
7. **Prover-only (NIZK) input (w):** This is a vector of values that supplements the input **x** for running the Program P. This input is accessible only by the Prover [see below].
8. **Output (P(x,w)):** This is a vector of M values that is produced by the Prover using inputs **x** and **w**.
9. **Proving Key (PK(P))**: This is a randomly generated key containing field elements generated by the **KeyGen** for facilitating proving.
10. **Verification Key (VK(P))**: This is a randomly generated key containing field elements generated by the **KeyGen** for facilitating proving.
11. **Proof (pi(P,x,w)):** This is a vector of field elements, specific to a Program
12. **KeyGen (P):** This is a machine that generates **PK(P)** and **VK(P)** and publishes it. This machine need not be too powerful.
13. **Input Provider (IP):** This is a machine that provides an input instance **x** and sometimes the additional input **w** to any machine via a private channel. This machine need not be too powerful.
14. **Prover:** This is a computation machine that takes QAP(P) / R1CS(P), Input **x**, Additional Input **w**, and Proving Key **PK(P)** to create [**P(x,w), pi(P,x,w)**]. This machine is normally considered powerful.
15. **Verifier:** This is a computation machine that takes **VK(P), QAP(P)**, input **x**, output

P(x,w) and Proof **pi(P,x,w).** It evaluates the proof to check that the computation was run correctly and outputs a boolean value.

Setup (Fixed: Program P, Prover Machine, Verifier Machine)

1. Create a program P represented in a high-level language **HLL(P).**
2. Use **COMP** to convert **HLL(P)** to **CIRC(P)**
3. Use **COMPCC** to convert **CIRC(P)** to **QAP(P)**
4. Create a sample verifier binary and do **KeyGen = (PK(P), VK(P))**
5. Transfer **QAP(P), PK(P)** to Prover and create Prover Binary.
6. If needed, Transfer **QAP(P), VK(P)** to verifier and create Verifier Binary.

Running an input (Proving)

1. **IP** provides an input **x** and transfers to Prover and Verifier.
2. **IP** provides an additional input **w** and transfers to Prover.
3. Use Prover Binary to do : **Prove(QAP(P), x, w, PK(P)) = [P(x), pi(P,x,w)]**
4. Prover sends **P(x), pi(P,x,w)** to Verifier.

Verification of input-run

1. The Verifier machine receives **x** from **IP,** and **P(x), pi(P,x,w)** from Prover
2. Use Verifier Binary to do : **Verify(QAP(P), x, P(x), pi(P,x,w), VK(P)) = {T | F}**
3. Send result and **P(x)** to **IP.**

System for implementing the above

1. The system uses a verifiable computing framework known as “Pequin” developed by an organisation “Pepper-Project” , provided under a BSD-style license by Authors at University of Texas, Austin and New York University
2. The system allows creating all the primitives as mentioned in the terminology
3. The system provides a compiler **COMP(HLL(P))** = **CIRC(P)** and also a compiler **COMPCC(CIRC(P)) = QAP(P)** . The compiler **COMP** supports ANSI C (A very big subset) and SFDL.
4. The system also provides creating Prover and Verifier Binaries with the following capabilites:
    1. Prover: prove, ExoInput (for additional input **w**)
    2. Verifier: KeyGen, GenInput, verify

Protocols and Libraries

The above system uses the following protocols and libraries.

1. Libsnark
2. Pinocchio Prove/Verify Protocol
3. Parsing Libraries for **COMP**

Actual Commands for the system

Current Working Directory : PEQUIN_HOME/pepper. All paths are relative to PEQUIN_HOME

1. Program is saved as program_name.c (or program_name.sfdl) in pepper/apps
2. For COMP[CC]: `make bin/program_name.params`
    1. This will create program_name.(f1index, params, pws, qap, qap.matrix_a, qap.matrix_b, qap.matrix_c) in pepper/bin/ folder
    2. This will create program_name.(c.defines, c.ZAATAR.circuit, c.ZAATAR.spec.cons, c.ZAATAR.spec.mem_consistency, c.ZAATAR.spec_temp, ctmp.c) in compiler/ folder
    3. This will create program_name.ZAATAR.spec in pepper/gen/ folder
3. For creating prover binary: `./pepper_compile_and_setup_P.sh program_name`
    1. This will create a binary pepper_prover_program_name in pepper/bin/ folder.
4. For creating verifier binary and run KeyGen: `./pepper_compile_and_setup_V.sh program_name program_name.vkey program_name.pkey`
    1. This will create a binary pepper_verifier_program_name in pepper/bin/ folder.
    2. This will create a verification key file program_name.vkey in pepper/verification_material/ folder
    3. This will create a proving key file program_name.pkey in pepper/proving_material/ folder
    4. **NOTE:** We can run `./bin/pepoer_verifier_program_name setup program_name.vkey program_name.pkey` for regenerating the key pair.
5. For generating inputs we have 2 options:
    1. Either we create a file (for instance program_name.inputs) in pepper/prover_verifier_shared/ folder.
    2. Or we can execute `./bin/pepper_verifier_program_name gen_input program_name.inputs` which will create a file program_name.inputs in

pepper/prover_verifier_shared/ folder.

3. The input format is unsigned integer elements in newlines. For more information, see section **Writing HLL Code for a program P**.

6. For running the prover to prove, we now write:
`./bin/pepper_prover_program_name prove program_name.pkey program_name.inputs program_name.outputs program_name.proof`

   1. This will run libsnark by reading the circuit from Prover Worksheet (pepper/bin/program_name.pws), the proving key (pepper/proving_material/program_name.pkey) and the input **x** from (pepper/prover_verifier_shared/program_name.inputs). For non-deterministic advice / hidden input **w**, see section **Additional Input and exo_compute()**
   2. This will create outputs file program_name.outputs and proof file program_name.proof and save in pepper/prover_verifier_shared/ folder.

7. For running the verifier to verify, we now write:

   `./bin/pepper_verifier_program_name verify program_name.vkey program_name.inputs program_name.outputs program_name.proof`

   This will run libsnark by reading the input program_name.inputs, the outputs program_name.outputs and the proof program_name.proof from (pepper/prover_verifier_shared/) folder. It also reads verification key (pepper/verification_material/program_name.vkey) and verifies the output and shows the result on STDIN.

Features

1. The compiler supports “data-dependant” loops by the Buffet System, but it requires LLVM and hasn't been thoroughly tested for correctness.
2. The circuit size is directly proportional to the program size in HLL (with all the loops and repeating constructs unrolled and all MACROs substituted).
3. The proving key and the verification key size are directly proportional to the complexity of the circuit.
4. The system utilizes fast math instructions on x86_64 architecture if available.
5. The proof size is always 288 Bytes (using pinocchio protocol)
6. Performance for verification only depends on the size of the input, and hence generally is constant.

Limitations

1. The system allows only a subset of C language for **COMP**.
2. The system doesn't support dynamic circuits [data-dependant sections of code or

loops]. It is compiled to a static circuit awaiting only the input

3. The system doesn't verify exo_compute (See section **Additional Input and exo_compute()**), it is treated as an input to the program, though it can be supplied data from inside the program. Non-determinism handles the theory.
4. The system works on honest-but-curious entities, all of which follow protocol but are untrusted.
5. Zero-knowledge guarantees are only on additional input **w** and this assumes that the prover **doesn't collude** with the verifier.

Writing HLL Code for a program P

The system provides circuit compiler for ANSI C and SFDL. Though SFDL is fully implemented, the C Compiler has certain limitations. Also, both the compilers require that the programs are written in a suitable format, which is described below.

C Programs:

1. The compiler supports a very large syntactical subset of the ANSI C grammar (leaving only function pointers and goto statement)
2. Only integral primitive types are supported throughout the program (char, int, unsigned, uint32_t, int32_t, uint64_t, int64_t, long). For using u<>_t types, need to include <stdint.h>
3. The input is restricted to unsigned 64-bit primitives (or any token that can be interpreted by GMP library)
4. The input in the program is encoded as a struct named **In,** and the inputs are populated in sequence as per C memory layout.
5. The output in the program is encoded as a struct named **Out,** and the outputs are printed in sequence as per C memory layout.
6. The main compute function has the signature **void compute(struct In*, struct Out*)**
7. No data dependant loops, early returns or data-dependant code segments are allowed and will give a compilation error.
8. No library calls are supported. All code needs to be inline and well-defined in entirety.

SFDL Programs

The same semantics apply to SFDL language, but no language features are missing. I haven't tested SFDL yet.

Additional Input and exo_compute()

Currently we can see that all the input is provided to the verifier. Also, the compiler doesn't support any library calls. We can overcome all this with the help of exo_compute().

The exo_compute() construct allows running of arbitrary binaries with user-provided input and allow their output to be interpreted by the calling program. **These programs are not part of the proof, as they are not part of the circuit, and only act as non-deterministic advisory input**. The binaries are stored as exo<NUMBER> in the pepper/bin/ folder. The interface between the programs are done through standard streams.

1. Hence exo_compute can be used for library calls.
2. Also, they can inject input that is unknown to the verifier and hence private to the prover.

Using exo_compute(): The Program Interface

exo_compute has the signature **void exo_compute(T** inputs, int* lengths, T* outputs, int exo_number)**. Here, **T =** Field Type, which can be treated as an integral type from the ones permissible in the program. The inputs are collected in a set of input vectors, and the array of these input vectors is given as the first argument. The lengths of the vectors (in-order) are in the array passed as second argument. The third argument provides the space to copy output of the exo-compute call. The last argument identifies the binary to be executed.

Using exo_compute(): The Binary Interface

The binary is provided input in 2 forms:

1. The number of output values expected is given as a command line argument.
2. The Input is provided on STDIN in the following format:

   <<number of input vectors>>

   <<number of elements in vector 1>> <<element 1,1>> <<element 1,2>> ...

   .

   .

   .
3. The output needs to be printed as whitespace separated tokens on STDOUT in a representation parsable by GMP library.