Libsnark Tutorial: Getting Started

Chan Nam Ngo
channam.ngo@unitn.it
University of Trento, Trento, Italy

November 14, 2018

1 Installation of the Tutorial Template

All the tutorials assume the Linux platform (e.g. Ubuntu). Students using other platforms should find equivalent alternatives. An easy solution would be VirtualBox[1] + Ubuntu[2].

Students are assumed to have a GitHub account and will be added with *read permission* into the CryptoFinTech2018Template repository. Students then can *fork* his/her own repository and *add Instructors as collaborators*. This provides secrecy for each student's own work, i.e. only the owner (Student) and the Instructors can read the code done by Student.

One should follow the instruction on `https://github.com/kurointrento/CryptoFinTech2018Template` for installation. The Tutorial Template includes libsnark for zk-SNARK and a starting point for students to do further complex tasks.

Other dependencies such as libssl (which also includes libcrypto) are also useful for some crypto tasks, e.g. random number generations, etc.

2 Introduction to the libsnark library

Details of the library can be found on the main repository page at `https://github.com/scipr-lab/libsnark`.

Basically libsnark is a library that provides a programming framework for zk-SNARK (Zero-Knowledge Succint Non-interactive Argument of Knowledge). Moreover, it includes two libraries gadgetlib1 and gadgetlib2 that have already implemented several common NP problems, e.g. the problem of finding the pre-image of a SHA-256 output. gadgetlib1 contains more useful functions while gadgetlib2 is simply a fancier refactorization of the first one.

We are going to follow the coding convention of libsnark, in particular gadgetlib1 for our tutorial and further student work.

[1] `https://www.virtualbox.org/wiki/Downloads`
[2] `https://www.ubuntu.com/`

In *some_gadget.hpp*, we shall pay attention to the *generate_r1cs_constraints* function and *generate_r1cs_witness_from_output* or *generate_r1cs_witness_from_input* functions. As an example, the *packing_gadget* is an arithmetic circuit that packs a bit vector into a single field element to compress the input. The *generate_r1cs_constraints* function generates the packing circuit and *generate_r1cs_witness_from_packed* takes the field element and decomposes it into bit vector while *generate_r1cs_witness_from_bits* do the reverse.

```
template<typename FieldT>
class packing_gadget : public gadget<FieldT> {
private:
/* no internal variables */
public:
const pb_linear_combination_array<FieldT> bits;
const pb_linear_combination<FieldT> packed;

packing_gadget(protoboard<FieldT> &pb,
const pb_linear_combination_array<FieldT> &bits,
const pb_linear_combination<FieldT> &packed,
const std::string &annotation_prefix="") :
gadget<FieldT>(pb, annotation_prefix), bits(bits),
packed(packed) {}

void generate_r1cs_constraints(const bool enforce_bitness);
/* adds constraint result = \sum  bits[i] * 2^i */

void generate_r1cs_witness_from_packed();
void generate_r1cs_witness_from_bits();
};
```

In the terms of the libsnark library, the arithmetic circuit is called a protoboard, the statement (the public values) is called primary input while the witness (the secret values) (e.g. the internal values of the circuit) is called auxiliary input.

3 Getting Started with the Tutorial Template

3.1 The CMakeLists

The first important file is CMakeLists.txt. To add a new test, e.g. *main_copy*, simply add the following lines into the CMakeLists.txt.

```
add_executable (
main_copy_test

tests/main_copy.cpp
```

```
utils/random.cpp
)
target_link_libraries (
main_copy_test

snark
gtest
)
```

3.2 The utils package

The utils folder provides functions to test proofs and generate randomness.

```
// Using the protoboard provided,
//this function generates a keypair, a proof and
// then verifies it.
template<typename ppT>
bool test_proof_from_protoboard
(const libsnark::protoboard<libff::Fr<ppT>> &pb);

// Using the protoboard provided,
//this function generates a proof, but tries to
// verify it with a witness that is completely set to 0.
template<typename ppT>
bool test_proof_wrong_witness_from_protoboard(
const libsnark::protoboard<libff::Fr<ppT>> &pb,
const libsnark::r1cs_primary_input<libff::Fr<ppT>> wrong_witness
);
```

```
RandomBitVectorGenerator(unsigned seed = 0) :
engine(seed), uniform_dist(0, 1) {};

libff::bit_vector generate_random_bit_vector(size_t width);
};
```

Students are not required to change this package unless they want to contribute back to the Tutorial Template.

3.3 The gadgets and tests packages

An example is provided for generating zk proof for a SHA-256 commitment.

```
// The constructore where all variables
// are declared and initialized
SHA256CommitmentGadget(libsnark::protoboard<FieldT> &pb,
const libsnark::pb_variable<FieldT> &value,
```

```
const libsnark::pb_variable_array<FieldT> &commitment,
const std::string &annotation_prefix);

// The function that generates the constraints,
// i.e. what is the expected computation,
// of the circuit
void generate_r1cs_constraints();

// The function that fills the circuit
// with all the input, output and intermediate values
void generate_r1cs_witness
(const libff::bit_vector& random_vector);
```

3.3.1 Constraints generation

If you look into the *generate_r1cs_constraints* function, pay attention to how the circuit is generated using gadgetlib1.

Firstly all variables should be initialized in the constructor.

```
// New variable allocation
this->padding.allocate(pb,
FMT(this->annotation_prefix, " padding"));

// Reuses value
auto value_digest = digest_variable<FieldT>(
pb, SHA256_digest_size, this->value.bits, padding,
FMT(this->annotation_prefix, " value_digest"));

// New variable allocation
auto hasher_output = digest_variable<FieldT>(
pb, SHA256_digest_size,
FMT(this->annotation_prefix, " hasher_output"));

// Connects value_digest, randomness (inputs) with hasher_output
this->sha256_hasher =
make_shared<sha256_two_to_one_hash_gadget<FieldT>>(
pb, value_digest, randomness, hasher_output,
FMT(this->annotation_prefix, " sha256_hasher"));

// Connects hasher_output with circuit output (this->commitment)
this->commitment_packer =
make_shared<multipacking_gadget<FieldT>>(
pb, hasher_output.bits, this->commitment, FieldT::capacity(),
FMT(this->annotation_prefix, " commitment_packer"));
```

Then the constraints are added.

```
value.generate_r1cs_constraints(true);
// Enforce bitness on value
// Constraints for value_digest not needed
// because of constraints for value
generate_r1cs_equals_const_constraint<FieldT>(this->pb, padding,
FieldT::zero(),
FMT(this->annotation_prefix, " padding_zero"));
randomness.generate_r1cs_constraints();
sha256_hasher->generate_r1cs_constraints();
// Constraints for hasher_output not needed
// because of constraints for commitment_packer
commitment_packer->generate_r1cs_constraints(true);
// true as to enforce bitness on commitment
```

Please be noted that the *multipacking_gadget* is useful for all of our tutorials as it compress the output (normally a SHA-256 hash) of many bits (256 bits) into a few Field elements (2 elements if Field size is 128 bits).

3.3.2 Adding witness

To fill a circuit, simply implements the *generate_r1cs_witness* function. Normally we just need to call the *generate_r1cs_witness* functions of the used subgadgets.

```
this->pb.val(padding) = FieldT::zero();
value.generate_r1cs_witness_from_packed();
randomness.generate_r1cs_witness(random_vector);
sha256_hasher->generate_r1cs_witness();
commitment_packer->generate_r1cs_witness_from_bits();
```

3.3.3 Tests

Then use the template to generate and test proof. The testing code was developed relying on the Google Test framework for C++ code. Each gadget is tested in a separate program and uses a fixture, common for all the tests in that program, in order to setup the environment.

All tests are run in the main function.

```
int main(int argc, char **argv) {
::testing::InitGoogleTest(&argc, argv);
return RUN_ALL_TESTS();
}
```

To initialize tests.

```
SHA256CommitmentTest() {
// General initialization
default_ec_pp::init_public_params();
pb = make_shared<protoboard<DefaultField>>();

// Circuit initialization
size_t commitment_size = div_ceil(SHA256_digest_size,
DefaultField::capacity());
// Commitment needs to be allocated before value,
// since it will
// be the primary input for the constraint system
commitment.allocate(*pb, commitment_size, "commitment");
pb->set_input_sizes(commitment_size);

value.allocate(*pb, "value");
sha256_commitment_gadget
= make_shared<SHA256CommitmentGadget<DefaultField>>(
*pb, value, commitment, "sha256_commitment_gadget");

// Constaints enforcement
sha256_commitment_gadget->generate_r1cs_constraints();
}
```

An example for a correct proof.

```
TEST_F(SHA256CommitmentTest,
WitnessSatisfiesCircuit) {
pb->val(value) = 0;

sha256_commitment_gadget->generate_r1cs_witness(
generator.generate_random_bit_vector(256));

EXPECT_TRUE(pb->is_satisfied());
}
```

And an example for a wrong proof.

```
TEST_F(SHA256CommitmentTest,
WrongCommitmentDoesNotSatisfyCircuit) {
pb->val(value) = 0;

sha256_commitment_gadget->generate_r1cs_witness(
generator.generate_random_bit_vector(256));

for (auto &elem : commitment) {
pb->val(elem) = 0;
```

```
}

EXPECT_FALSE(pb->is_satisfied());
}
```

4 Getting Started

Make a copy of the SHA256Commitment (gadget and tests). Solution is provided in the original repository, `https://github.com/kurointrento/CryptoFinTech2018Template`.

See the changes in commit `https://github.com/kurointrento/CryptoFinTech2018Template/commit/c0f202b0155cb45c1b9093c26c12d6780d0feddb`.

Acknowledgement

This template is derived from the work done by Elia Geretto [3], in the Research Project Course (Fall 2017), advised by Prof. Fabio Massacci [4] and TA Chan Nam Ngo [5].

[3] *elia.geretto@studenti.unitn.it*
[4] *fabio.massacci@unitn.it*
[5] *channam.ngo@unitn.it*