`luigi-qif` Documentation

The TAMBA team at Galois

March 2020

1 Quick-start

Inside the docker container (from the `/luigi-qif` directory), run:

- `./luigi-qif.py --priors examples/sum.priors --mpc examples/sum.mpc static-leakage`
- `./luigi-qif.py --priors examples/sum.priors --mpc examples/sum.mpc 3 dynamic-leakage`

For information on how to interpret the resulting output of the `static-leakage` and `dynamic-leakage` commands, see Section 3.b and Section 3.c, respectively.

2 Introduction

Multi-Party-Computation technology is employed because the parties involved in a shared computation would like to keep their inputs to that computation private. MPC protocols have proofs which show that the parties involved in the computation should learn nothing aside from the result of the computation. This fact, however, is insufficient to prevent a party from learning (something about) the private input.

For example, consider an MPC program $f(a, b) = a + b$, where input a is provided by party A and input b is provided by party B. Even if f is executed completely securely by the MPC protocol, after the protocol executes, party A learns the resulting sum, S. Because party A knows its own input, a, it can recover party B's input since $b = S - a$.

The goal of the `luigi-qif` tool is to help the user understand how much information a given computation leaks.

3 Basic Usage

```
Usage: luigi-qif.py [OPTIONS] COMMAND [ARGS]...

Options:
  --priors TEXT                    The new-line separated priors file
                                   [required]
  --mpc TEXT                      The MPC algorithm's source code  [required]
  --model-counter [BSat|ApproxMC]
                                  which method to use to compute model counts
  --help                          Show this message and exit.

Commands:
  dynamic-leakage
  static-leakage
```

3.a Options

MPC File

The MPC file is Python code which specifies the MPC program to execute. See the section "Python API" below.

Prior File

The job of the prior file is to describe the context that the adversary has about the situation. Specifically, the prior file holds the minimum and maximum value the adversary might guess for each piece of private data. This helps our tool reason about whether a result would improve the adversary's guess or not. For example, a computation with a result that implies the input was zero or greater "rules out" many possible inputs (all the negative values). But if the adversary knows the input is the (positive) mass of an object then we specify this in the prior file, and `luigi-qif` computes leakage with respect to only things the adversary did not already know to be true. In this case, ruling out negative values does not increase the leakage.

Formally, the `luigi-qif` tool assumes a prior distribution on private data where each private input is independent of every other private input, and is uniformly distributed on some finite integer range. The prior file will specify that range.

The actual syntax of the prior file looks something like this:

```
0 4
0 2
```

A prior file consists of a list of (UNIX) newline-separated lines, each of which corresponds to a private input to the MPC program. The first integer specifies the inclusive lower-bound on the uniform prior distribution of the private input, and the second integer specified the inclusive upper-bound. The *i*-th line of the prior file is used as the prior for the *i*-th private input (where private inputs are ordered in the order that they are called in the MPC program). It is valid to have extra lines/entries in the prior file; they will be ignored.

3.b Static Leakage

“Static” mode is one of two possible modes the `luigi-qif` tool can operate in. Static leakage, also known, as predictive leakage, asks the question: “how much information do we *expect* will be leaked?” Static leakage is agnostic to any concrete input or output to the MPC program.

The following command will output the expected min-entropy leakage of a MPC program which takes 10 private inputs, each an integer with value in the range of 0 to 4 (inclusive), and produces the sum of the inputs:

Example Run

```
$ ./luigi-qif.py --priors examples/sum.priors --mpc examples/sum.mpc static-leakage
Number of possible outputs: 41
STATIC LEAKAGE: 5.357552
```

The “count=. . . ” line is debugging/progress information that is emitted during the counting process. The count 41 means that the expected min-entropy leakage is $\log_2(41) = 5.357552$.

3.c Dynamic Leakage

“Dynamic” mode is the other mode that the `luigi-qif` tool can operate in. Unlike the static leakage mode, the dynamic leakage mode asks: “how much information *was* leaked in a *concrete* execution of the MPC program?” As a result, the dynamic leakage operation mode takes, as arguments, the concrete outputs of the MPC program.

The dynamic leakage mode outputs the prior vulnerability, the posterior vulnerability, and the min-entropy leakage.

Example Run

```
$ ./luigi-qif.py --priors examples/sum.priors --mpc examples/sum.mpc dynamic-leakage 3
Prior vulnerability: 1/9765625
Posterior vulnerability: 1/220
Leakage: 15.437921
```

In the above example, the argument “3” is a concrete output that was seen as the result of an execution of the MPC program. The “count=. . . ” are progress/debugging output.

Model Counter

For details on the trade-offs between the model counters, see the section “Different Model Counters” below.

4 Python API

The Python API is used to specify an MPC program. It's largely compatible with the SPDZ/MP-SPDZ/SCALE-MAMBA Python API. As a result, many MPC programs can be analyzed without modification by `luigi-qif`.

4.a Sum Example

```python
# This example describes an MPC program which takes integers from several parties,
# and prints out the sum of these integers.

# Construct a list of ten inputs. The argument to the get_input_from function is
# which party to take input from. The order is the order in the prior file.
numbers = [sint.get_input_from(i) for i in range(10)]

# This makes a constant. The use of the sint constructor is optional.
# That is, it would've been okay to just write: acu = 0
acu = sint(0)
# Then we add the numbers
for x in numbers: acu += x

# Print the result (when the MPC program is run), not at analysis time.
print_ln("RESULT: %s", acu.reveal())
```

The above example is a valid SPDZ program. One thing to note about the above example is the use of `print_ln` instead of `print`. The `print` function, itself, would only cause output to be printed at compile/analysis-time. It would have no effect on the actual runtime execution of the MPC program.

5 Different Model Counters

`luigi-qif` supports two different model counting techniques, each of which has different trade-offs:

BSat (bounded-sat; the default model counter) uses the Z3 SMT solver to enumerate all the solutions. As a result, it is best-suited for smaller models with small counts (under one million is a good rule of thumb). BSat will always return an exact result.

ApproxMC uses the ApproxMC model counter. It is an approximate model counter. As a result, it will return an approximate result. It is well-suited for models with large counts.

It is safe to run ApproxMC on models with small counts. However, BSat might return a more precise answer (since BSat always returns the exact answer). In addition, BSat might be faster (since an input formula must be pre-processed before it can be run through ApproxMC).

6 Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA).

The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.