Reviews:

===========================================================================

S&P 2018 Review #290A

---------------------------------------------------------------------------

Paper #290: EzPC: Programmable, Efficient, and Scalable Secure Two-Party

Computation

---------------------------------------------------------------------------

Overall merit: 4. Weak accept - While flawed, the

paper has merit and we should

consider accepting it.

===== Brief paper summary (2-3 sentences) =====

Two-party computation (2PC) protocols are used when two parties who do not fully trust each other want to compute an output together such that they do not learn each other's input. The problem with such protocols is that they are often troublesome to implement. Compilers exist to create such protocols in either boolean or arithmetic circuits, but not a combination of both without a lot of complexity. The ABY framework from [1] does this, but it is quite low-level. This paper suggests EZPC1, a framework that generates efficient and scalable 2PC protocols from high-level programs, such that the implementer does not need to worry about how the program works, but only what it needs to do.

===== Strengths =====

The paper introduces the problem and unique contributions very clearly in the introduction. They use a running example of a classier to demonstrate their contributions throughout the paper.

They provie formal correctness and security theorems for their compiler in the case of an honest-but-curious adversary, reducing the security argument to a security argument of their backend (ABY, [1])

The performance of their compiled code is comparable (j some cases x19) faster than hand-crafted state-of-the-art implementations.

They demonstrate that it is practical for a non-cryptographer by having a non-cryptographer write the first 2PC implementation of the BONSAI protocol using their compiler.

===== Weaknesses =====

Justify with examples how many mistakes are made without it.

Comparison of speed of compiled code to other works, particularly to [1].

Are there many situations where a programmer does not know much cryptography but wants to implement a secure 2PC protocol with a combination of boolean and arithmetic circuits, not using the ABY framework?

===== Detailed comments for the author(s) =====

I would like to understand more clearly how exactly the compiler works. In particular, how it can be "cryptographically cost aware"?

In related work, I see that other compilers do not use a combination of arithmetic and boolean circuits. However, it would be interesting a see a more complete comparison of benchmarks between this compiler and those. In particular with regards to efficiency of certain implementations of protocols.

I see the security reduction reduces the security to the backend ABY. Does this apply to things such as side-channels?

While I believe the authors that theorem 2 can be extended to consider more malicious adversaries (although in the conclusion the phrase "malicious security" I think should be changed), it may be best not to claim those things if they are not actually proved.

Some typos here and there. E.g. ""the largest compute that can be done…"

[1]. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation.]

===========================================================================

S&P 2018 Review #290B

---------------------------------------------------------------------------

Paper #290: EzPC: Programmable, Efficient, and Scalable Secure Two-Party

Computation

---------------------------------------------------------------------------

Overall merit: 3. Weak reject - The paper has flaws,

but I will not argue against it.

===== Brief paper summary (2-3 sentences) =====

The paper presents EzPC, a high-level language for secure 2-party computation. EzPC generates protocols that combine arithmetic and boolean circuits, and develops a code partitioning method which guarantees scalability. Practical evaluation suggests that EzPC achieves good efficiency compared to prior work.

===== Strengths =====

- Developing high-level 2PC languages and compilers that target

combinations of arithmetic and boolean circuits is a useful

improvement over prior work

- Practical evaluation demonstrates good benefits

===== Weaknesses =====

- Formal guarantees are not available. The main definitions and proofs

should have appeared in an appendix, not in an anonymized citation

- Section 4 is difficult to follow

===== Detailed comments for the author(s) =====

EzPC is a high-level language for writing 2PC programs in a familiar framework of high-level programming. EzPC programs are compiled into combinations of arithmetic and boolean circuits. The choice between arithmetic and boolean circuits is delegated to a cost-aware compiler which decides the best strategy, and inserts the necessary conversions between the two representations. Practical evaluation shows that the generated code behaves well wrt hand-written code.

The paper pursues an interesting line of research, but suffers from a number of shortcomings:

- programming language: the type system is restricted to conditional

statements that have public guards. However, it is often required,

primarily for efficiency reasons, to branch on secret values. The

common strategy is to declassify secret values before branching.

The paper does not consider this case. I do not consider that the

paper should address this issue to be accepted, but it should at

least mention the problem. However, forbidding high branches

significantly simplifies the formal treatment of the paper. In

particular, the proof of Theorem 2 is direct.

- secure code partitioning. This section is very terse and difficult

to follow. Providing an example of partitioning would help.

- implementation: a number of points deserve clarification. For

instance, the implementation currently adopts a very simple type

inference strategy; does it lead to rejecting secure programs?

Moreover, the compiler performs several optimizations (CSE, etc);

are we guaranteed that the formal results still apply?

On the positive side, I found the benchmarks convincing.

Overall, I think that the paper pursues an interesting and potentially fruitful direction. Nevertheless, I am reluctant to support acceptance in view of the limitations discussed above.

===========================================================================

S&P 2018 Review #290C

---------------------------------------------------------------------------

Paper #290: EzPC: Programmable, Efficient, and Scalable Secure Two-Party

Computation

---------------------------------------------------------------------------

Overall merit: 3. Weak reject - The paper has flaws,

but I will not argue against it.

===== Brief paper summary (2-3 sentences) =====

This paper builds on the ABY secure computation framework that includes support for both arithmetic and boolean circuits. While that work provide low-level primitives, this work provides higher level constructions that, importantly, allow for the use of either arithmetic or boolean circuits depending on the parameters of the problem to be solved. The language is built to resemble imperative languages such as C. Proofs of security and correctness are given and the compiler is evaluated against other work with a focus on privacy-preserving machine learning algorithms.

===== Strengths =====

- New compiler that automatically picks between arithmetic and boolean circuits and designs the code to interface between them.

- Formal security guarantees.

- Evaluation against machine learning algorithms.

===== Weaknesses =====

- No discussion of compiler validation.

- Secure code partitioning requires manual work by the programmer.

- Evaluation against other competing frameworks is not done on a consistent platform.

===== Detailed comments for the author(s) =====

I appreciated that the work focused on how to make the job of the programmer easier. This seems like the next natural direction for secure computation compilers to go, to consider how to optimize existing protocol frameworks to provide the right circuits depending on the scenario. The innovation here is the cost-aware compiler that automatically picks the right representation for various parts of the computation and compiles them together. That's very interesting.

The formal development using big-step semantics is welcome to see and is discussed in detail. What is not discussed, though, is how this maps to the implementation. How did you ensure that your Python implementation of EzPC preserved all of the big-step semantics described by your ladder diagrams? What type of validation was done on the compiler to ensure its correctness? This is often an area where implementations of secure computation compilers can go somewhat awry. The authors should look at the paper "Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation" (Mood et al., Euro S&P 2016) and discuss how validation occurs in this work (that paper should also be cited here).

While semi-honest models are the most performant, a number of secure computation frameworks provide support against covert or malicious adversaries. It would be worth discussing here what would be required for this framework to provide support against those adversaries.

The partitioning algorithm is interesting and the idea of the compiler decomposing a program into sequences that preserve the memory space while maintaining security and correctness guarantees is something that could be widely applicable - however, its utility is limited by the fact that the programmer needs to decompose the program into a sequence of smaller programs themself.

With regards to the evaluation, it appears that the authors did not run other solutions on their benchmarking platforms. Therefore, the results that are provided in the evaluation are not apples-to-apples comparisons. For example, the Bost et al. benchmarks are run on machines with Core-i7 processors with 8 GB of RAM and are run on a LAN where the latency is artificially lengthened to 40 ms to simulate a LAN. Meanwhile, the benchmarks in this work are performed on machines with Xeon processors with 28 GB of RAM, so it's difficult to effectively compare processing time. In addition, comparing LAN time against this other work does not provide a decent comparison. Moreover, the latter benchmarks in this work, considering Tenserflow and Bonsai, are not ported to other platforms for comparison. Did the authors of this work try to get the code from the other groups so that consistent benchmarking can be done? This has become the standard way of doing performance evaluation in this communit y so if there were reasons why this was not feasible, those should be clearly stated.

On a related note, perhaps it is because of the lack of visibility into the other compilers but when the authors compare their results to MiniONN, they sound almost mystified by their own results - as they point out, these results are surprising in terms of how well ExPC does given that MiniONN "has a much more efficient preprocessing phase and has SIMD capabilities that allow it to perform matrix operations efficiently". This would be a good opportunity for the authors to analyze exactly what it is that makes EzPC better in this circumstance through an in-depth analysis of each step of the computation. That would be beneficial as it could point to where research should be focused to ensure that we are optimizing on the right things.

The paper was well-written and easy to follow. I particularly appreciated the bibliography with links back to the pages where the references occurred. This would be very beneficial to have as a standard for published papers.

========================================================================