Usenix

1. Language is less expressive than various others
   * No public input type (except hardcoding in the code)
   * No secret array indices
2. Compare with other compilers like CBMC-GC, PICCO
   * Though only Boolean compare size of circuits produced
3. Cryptographic cost aware – Novelty and non-triviality
   * Mixed mode decisions are trivial + no justifications
   * Examples are simple and no evidence for optimal choice
   * Argue why the proposed implementation is optimized?
   * Would like more discussion on cryptographic cost aware compiler. Thinks that there is a default choice for each basic op. Missed that add has multiple options. Also, complains that choices are one size fits all; not depend on network and other parameters
   * Specify cost of conversions to better understand above justification
4. Not convinced that Yao slower
   * Groce et al had better performance
5. Partitioning
   * Simple and not novel; Yao has same idea
   * Not cite some papers Groce et al., Mood et al., Yao pipelining
   * Why do partitioning only using arithmetic values?
6. Performance results not good;
   * Groce had better numbers
   * Comm only marginally better compared to previous works
7. Evaluation
   * Is there a bandwidth limit in LAN or WAN setting?
   * More detail on ABY default mode
8. Other comments
   * Use less math in Theorem 1 to make it more readable
   * Why another DSL? Could this be built over other existing DSLs? Argue how the language is superior to others in literature
     1. Cost aware compilation in this paper could have been integrated with compilers with less effort

Papers

1. Groce et al. <https://eprint.iacr.org/2016/458.pdf>
2. Mood et al. <https://arxiv.org/pdf/1506.02954.pdf>
3. Mood et al. Frigate: validated compiler

Comments:

1. I don’t think that Groce et al. or Mood et al. do pipelining. Groce et al. could not do Naïve Bayes for Audiology because circuit does not fit into memory and they didn’t have pipelining from disk. Mood et al. only cites Yao pipelining in related work.

Oakland

1. How was it to write programs directly in ABY?
2. How does performance compare to ABY code?
3. Are there many situations where programmer uses 2PC not knowing crypto and can’t use ABY?
4. Explain cryptographic cost aware compiler
5. Compare some benchmarks with other compilers even though either full Boolean or Arithmetic
6. Security reduces to ABY; does this apply to sidechannels?
7. Main definitions and proofs should appear in appendix and not anonymous citation
8. Partitioning
   1. Hard to follow
   2. Example would help
9. Language
   1. Branching on secret values
   2. Declassifying intermediate results
   3. Simple type inference; does it reject correct secure programs?
   4. Many optimizations like CSE, etc; do formal guarantees still apply?
10. Compiler
    1. Compiler validation; is python implementation faithful to big step semantics provided?
    2. Look at Frigate: Mood et al Euro S&P 2016 for how validation occurs
11. Partitioning is manual
12. Evaluation not done on a consistent platform
    1. Bost et al and our LAN network as well as machine configs is different
    2. Try porting Tensorflow benchmarks and Bonsai to other frameworks
    3. Ask for code of others; if this cannot be done for some reason, please cite those reasons
    4. Should not be surprised by own results; should analyse to see why their results are better
13. Cryptography
    1. Why not malicious or covert security? What is needed for that

Usenix Rebuttal

We thank the reviewers for their comments. We structure the rebuttal to address main concerns in the first ~500 words, leaving the rest to a detailed section.

Comparison with Groce et al., CBMC-GC, Yao-only etc.:

We thank Reviewer A for the Groce et al. reference. Our numbers should not be directly compared with theirs since they report only the "online" time of the protocols, while we report the sum of online and "offline" times. Further, the benchmarks in Groce et al. are quite small, and so, it is not surprising that the Yao-only approach works well. It is unclear that their approach would scale to our larger benchmarks, e.g. DNNs or matrix factorization (more thorough comparison below).

CBMC-GC takes 30seconds and generates 1.8 million gates to multiply 2 5x5 matrices. On the other hand, MNIST logistic regression (Table 5 in our paper) consists of multiplying 2 192x192 matrices -- and EzPC generated protocol takes 0.7 seconds (in WAN) with 35k gates.

As for the Yao-only approach in general, we are not the first to observe that it does not scale well beyond examples with small number of multiplications ([18], [26], [31], [40], [37]), especially for large-scale arithmetic computations commonly found in Machine Learning (ML) algorithms -- the main use case of EzPC. Indeed, this observation is one of the main drivers behind development of the specialized MPC protocols in previous work ([40], [37]). Therefore, it is critical to combine the arithmetic and boolean circuits for the kind of ML algorithms we apply EzPC to.

Performance gains:

For large benchmarks, such as CIFAR 10 (Table 7) and Matrix Factorization (Table 8), EzPC performs much better than the previous best known algorithms (~2x and ~19x resp.). Unsurprisingly, for small benchmarks in Tables 1-4, where the baseline time itself is only ~5seconds, the performance gains of EzPC do not seem as impressive. But even then we would like to point out that while most of the previous work on those benchmarks developed specialized protocols, EzPC framework is general, programmer friendly, and does not require any crypto expertise.

Language features:

EzPC language itself is quite simple and currently does not support features such as secret indices and public inputs. However, the language design has been driven by machine learning, specifically secure prediction case studies. As we show, even with such a simple language, we can implement a lot of such examples.

We also remark that while the source language is simple, EzPC compiler is the first compiler that generates a mix of arithmetic and Boolean circuits, while hiding all the low-level crypto details from the programmer.

Cost-awareness:

Our compiler currently applies certain heuristics to choose arithmetic or boolean circuits for different parts of the program (Sec. 5). Even with such simple heuristics, we are able to achieve the demonstrated efficiency and scalability. A more precise cost model that factors in the cost of network, input sizes, etc. would only improve performance (more on it below).

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

Detailed responses:

Comparison with Groce et al.:

Groce et al. improves on online time and online communication of native Yao's garbled circuits by moving more computation to the offline (input independent) phase. However, the overall times remain roughly the same. As is standard, we are interested in the overall execution times of our protocols -- not just the online phase. Additionally, the benchmarks provided in Groce et al. (and also Bost et al.) are small "warm-up" benchmarks that we provide in our paper for completeness (vector dimensions for inner product are 30 and 47). The full power of our techniques can be best seen in the ease of running large benchmarks such as logistic regression, and neural networks. These require much larger number of multiplications and no one has attempted to implement these using garbled circuits. For instance, simplest benchmark of logistic regression requires 10 inner products of length 784. The Boolean circuit for one inner product itself would roughly consist of 800k gates (whereas the number of gates in the hybrid approach for the entire computation is only 35k). The work of Groce et al. can potentially be used as a backend for the Boolean part of the compute in our protocols - we leave this exploration to future work.

Cryptographic Cost-aware compiler:

We do not claim that our compiler choices are optimal. Multiplication is always done using arithmetic secret sharing as the cost of multiplication in Boolean shares is much higher (for 64-bit integer multiplication, using arithmetic shares, we need to do only 2 multiplications of uint64. Over Yao, the circuit size would be 64\*64 and would require at least 4096 AES operations in the online phase.). Addition \*does not\* have a default way of evaluation and is done either over Boolean or Arithmetic shares depending on available shares of operands.

Secure code partitioning:

Our technique of secure code partitioning, although not entirely new, provides a generic way to execute large programs without going into the specific details of cryptographic protocol. We describe it only in the arithmetic setting but as the reviewer correctly points out, same can be done in Boolean setting as well.

Secure array accesses:

Combining the techniques of EzPC and ObliVM to support Boolean, arithmetic and secret array access would be an interesting future direction. However, for ML algorithms such as logistic regression, linear classifier, DNNs/CNNs, the access patterns are uniform and so providing for secret array accesses will not improve performance.

Integrating in an existing compiler:

We found that we can write many secure prediction algorithms in an EzPC-like simple language, without pulling in complexities of other languages and systems. When we add more features to EzPC, such as ORAM support, we could definitely consider the option of integrating with some other compiler.

Final Rebuttal

We thank the reviewers for their comments. We structure the rebuttal to address main concerns in the first ~500 words, leaving the rest to a detailed section.

Comparison with Groce et al., CBMC-GC, Yao-only etc.:

We thank Reviewer A for the Groce et al. reference. Our numbers should not be directly compared with theirs since they report only the "online" time of the protocols, while we report the sum of online and "offline" times. Further, the benchmarks in Groce et al. are quite small, and so, it is not surprising that the Yao-only approach works well. It is unclear that their approach would scale to our larger benchmarks, e.g. DNNs or matrix factorization (more thorough comparison below).

We ran CBMC-GC and it timed out after five hours in multiplying two 20x20 matrices. Our DNNs are much bigger, for instance, CIFAR requires multiplying a 64x576 matrix with a 576x1024 matrix.

As for the Yao-only approach in general, we are not the first to observe that it does not scale well beyond examples with small number of multiplications ([18], [26], [31], [40], [37]), especially for large-scale arithmetic computations commonly found in Machine Learning (ML) algorithms -- the main use case of EzPC. Indeed, this observation is one of the main drivers behind development of the specialized MPC protocols in previous work ([40], [37]). Therefore, it is critical to combine the arithmetic and boolean circuits for the kind of ML algorithms we apply EzPC to.

Performance gains:

For large benchmarks, such as CIFAR 10 (Table 7) and Matrix Factorization (Table 8), EzPC performs much better than the previous best known algorithms (~2x and ~19x resp.). Unsurprisingly, for small benchmarks in Tables 1-4, where the baseline time itself is only ~5seconds, the performance gains of EzPC do not seem as impressive. But even then we would like to point out that while most of the previous work on those benchmarks developed specialized protocols, EzPC framework is general, programmer friendly, and does not require any crypto expertise.

Language features:

EzPC language itself is quite simple and currently does not support features such as secret indices and public inputs. However, the language design has been driven by machine learning, specifically secure prediction case studies. As we show, even with such a simple language, we can implement a lot of such examples.

We also remark that while the source language is simple, EzPC compiler is the first compiler that generates a mix of arithmetic and boolean circuits, while hiding all the low-level crypto details from the programmer.

Cost-awareness:

Our compiler currently applies certain heuristics to choose arithmetic or boolean circuits for different parts of the program (Sec. 5). Even with such simple heuristics, we are able to achieve the demonstrated efficiency and scalability. A more precise cost model that factors in the cost of network, input sizes, etc. would only improve performance (more on it below).

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

Detailed responses:

Comparison with Groce et al.:

Groce et al. improves on online time and online communication of native Yao's garbled circuits by moving more computation to the offline (input independent) phase. However, the overall times remain roughly the same. As is standard, we are interested in the overall execution times of our protocols -- not just the online phase. Additionally, the benchmarks provided in Groce et al. (and also Bost et al.) are small "warm-up" benchmarks that we provide in our paper for completeness (vector dimensions for inner product are 30 and 47). The full power of our techniques can be best seen in the ease of running large benchmarks such as logistic regression, and neural networks. These require much larger number of multiplications and no one has attempted to implement these using garbled circuits. For instance, simplest benchmark of logistic regression requires 10 inner products of length 784. The Boolean circuit for one inner product itself would roughly consist of 800k gates (whereas the number of gates in the hybrid approach for the entire computation is only 35k). The work of Groce et al. can potentially be used as a backend for the Boolean part of the compute in our protocols - we leave this exploration to future work.

Cryptographic Cost-aware compiler:

We do not claim that our compiler choices are optimal. Multiplication is always done using arithmetic secret sharing as the cost of multiplication in Boolean shares is much higher (for 64-bit integer multiplication, using arithmetic shares, we need to do only 2 multiplications of uint64. Over Yao, the circuit size would be 64\*64 and would require at least 4096 AES operations in the online phase.). Addition *does not* have a default way of evaluation and is done either over Boolean or Arithmetic shares depending on available shares of operands.

Secure code partitioning:

Our technique of secure code partitioning, although not entirely new, provides a generic way to execute large programs without going into the specific details of cryptographic protocol. We describe it only in the arithmetic setting but as the reviewer correctly points out, same can be done in Boolean setting as well.

Secure array accesses:

Combining the techniques of EzPC and ObliVM to support Boolean, arithmetic and secret array access would be an interesting future direction. However, for ML algorithms such as logistic regression, linear classifier, DNNs/CNNs, the access patterns are uniform and so providing for secret array accesses will not improve performance.

Integrating in an existing compiler:

We found that we can write many secure prediction algorithms in an EzPC-like simple language, without pulling in complexities of other languages and systems. When we add more features to EzPC, such as ORAM support, we could definitely consider the option of integrating with some other compiler.