Verified Computation via Compilation to Abstract Machines

Ryan Estes
restes@uvm.edu
University of Vermont

David Darais
darais@galois.com
Galois, Inc.

Joseph P. Near
jnear@uvm.edu
University of Vermont

ABSTRACT

Verified computation techniques enable the construction of zero-knowledge proofs that an output was computed by a particular program—without revealing the program's input. Existing techniques work by converting programs into circuits or constraints, which makes dealing with control flow challenging. We present a new approach that compiles the program to a specialized abstract machine that contains no control flow and is particularly well-suited to zero-knowledge proof.

1 INTRODUCTION

Verifiable computing [3] has the goal of offloading computation to an untrusted third party, while maintaining the ability to verify that the computation has been performed correctly. Techniques for verifiable computing often rely on zero-knowledge proofs [4], which allow a *prover* to convince a *verifier* that the prover knows a (secret) witness w such that $f(w) = o$—without revealing w.

Zero-knowledge protocols typically require converting the function f to a circuit or set of constraints. Two main approaches have been proposed for this: (1) compile the program into a circuit [5], or (2) manually design a circuit that implements a CPU with a limited instruction set, and compile the program into instructions for this CPU [1]. Option (1) makes control flow challenging—since circuits do not support control flow natively—and option (2) often results in circuits that are too large for real programs.

We present a new approach that combines the best properties of both options. Our approach compiles a program written in a high-level language called WizPL into a specialized *abstract machine* whose set of operations is tailored to the program. The resulting abstract machine has *no control flow*, so it can be efficiently represented in zero-knowledge proof systems.

2 THE WIZPL COMPILER

The architecture of our compiler appears in Figure 1. The goal of this architecture is to produce an abstract machine for an input program with the smallest possible set of operations, while also keeping each operation simple and minimizing the number of steps needed to execute the program.

Most of the steps in our compiler are standard—we first perform a continuation-passing style transformation, then a closure conversion step. The third step—defunctionalization [2]—is not used in compilers for real hardware. This step transforms *functions* into *data*, effectively encoding each of the program's functions as an operation type in the abstract machine. The example WizPL program below implements the factorial function, and runs it on a secret input of 5 (the `secret` keyword denotes witness values).

```
fact n = if n == 0            w = secret 5
  then 1                      main () = fun () => fact w
  else n * (fact (n - 1))
```

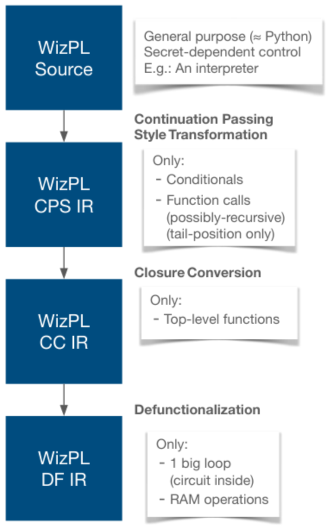

Figure 1: Architecture of the WizPL Compiler

The WizPL compiler transforms the factorial program into an abstract machine with 6 operations; the complete abstract machine, written as a C program, is available in Figure 2 in the Appendix.

The resulting abstract machine has no control flow, but relies heavily on random-access memory (encoded using `store` and `load` operations in the code of Figure 2). RAM operations are encoded in zero-knowledge proof systems by recording a complete memory trace of the program's execution, and constructing *consistency checks* in the circuit representation of the program that ensure reads and writes are consistent with the memory trace.

Since WizPL programs are functional, our abstract machines never re-use memory locations, and a memory *trace* in our setting can be built by simply dumping the contents of memory at the end of the program's execution. This simplifies the encoding of memory operations in the zero-knowledge proof system, since we only need *read-only memory*.

3 CONCLUSION & OPEN CHALLENGES

Our approach enables compiling high-level programs with control flow, recursion, and user-defined data structures into abstract machines that can be efficiently represented in zero-knowledge proof systems. Our initial results suggest that this approach is more efficient than either of the two options mentioned in the introduction.

We believe that significant improvements can be made by optimizing the tradeoff between the *number* and *complexity* of operations. The complete set of operations must be encoded in the resulting circuit; more complex operations results in a larger circuit for each step of the abstract machine's execution, but potentially allows the machine to finish in fewer steps; exploring this tradeoff remains future work.

```
for (unsigned int i = 0; i < 19; ++i) {
  int tag = load(state_f) + (case_id * 65536);
  case_id = 0;
  switch (tag) {
    case O_F2:
      new_f = store(O_halt);
      new_a0 = state_a0;
      break;
    case O_F0:
      T0 = load(state_f + 2) * state_a0;
      new_f = load(state_f + 1);
      new_a0 = T0;
      break;
    case 65536:
      T0 = store(O_F0);
      store(state_a1);
      store(state_a2);
      T1 = state_a2 - 1;
      new_f = state_a0;
      new_a0 = T1;
      new_a1 = T0;
      break;
    case 65537:
      new_f = state_a1;
      new_a0 = 1;
      break;
    case O_F1:
      T0 = state_a0 == 0;
      new_f = store(T0);
      case_id = 1;
      new_a0 = load(state_f + 1);
      new_a1 = state_a1;
      new_a2 = state_a0;
      break;
    case O_main:
      T0 = store(O_F1);
      store(T0);
      T1 = store(O_F2);
      new_f = T0;
      new_a0 = W_0;
      new_a1 = T1;
      break;
    }

    state_f = new_f;
    state_a0 = new_a0;
    state_a1 = new_a1;
    state_a2 = new_a2;
}

int result = state_a0;
```

Figure 2: Complete abstract machine for factorial.

REFERENCES

[1] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, pages 90–108. Springer, 2013.

[2] Olivier Danvy and Lasse R Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174, 2001.

[3] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*, pages 465–482. Springer, 2010.

[4] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728, 1991.

[5] Riad S Wahby, Srinath TV Setty, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. In *NDSS*, 2015.