Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation

Benjamin Mood*, Debayan Gupta†, Henry Carter‡, Kevin R. B. Butler* and Patrick Traynor*
*University of Florida
†Yale University
‡Georgia Institute of Technology

Abstract—**Recent developments in secure computation have led to significant improvements in efficiency and functionality. These efforts created compilers that form the backbone of practical secure computation research. Unfortunately, many of the artifacts that are being used to demonstrate new research for secure computation are incomplete, incorrect, or unstable, leading to demonstrably erroneous results and inefficiencies - extending even to the most recently developed compiler systems. This is a problem because it hampers research and undermines feasibility tests when other researchers attempt to use these tools. We address these problems and present Frigate, a principled compiler and fast circuit interpreter for secure computation. To ensure correctness we apply best practices for compiler design and development, including the use of standard data structures, helpful negative results, and structured validation testing. Our systematic validation tests include checks on the internal compiler state, combinations of operators, and edge cases based on widely used techniques and errors we have observed in other work. This produces a compiler that builds correct circuits, is efficient and extensible. Frigate creates circuits with gate counts comparable to previous work, but does so with compile time speedups as high as 447x compared with the best results from previous work in circuit compilers. By creating a validated tool, our compiler will allow future secure computation implementations to be developed quickly and correctly.**

1. Introduction

Secure Multiparty Computation (SMC) has long been regarded as a theoretical curiosity. First proposed by Yao [55], SMC allows two or more parties to compute the result of a function without exposing their inputs. The identification of such primitives was groundbreaking, creating opportunities by which untrusting participants could calculate results of mutual interest without requiring all individuals to identify a mutually trusted third party. Unfortunately, it would take more than 20 years before the creation of the first SMC compiler, Fairplay [36], demonstrated that these heavyweight techniques were remotely practical.

The creation of the Fairplay compiler ignited the research community. In the following decade, SMC compilers improved performance by multiple orders of magnitude, significantly reduced bandwidth overhead, and allowed for the generation and execution of circuits composed of tens of billions of gates [38], [31], [22], [30]. These efforts have brought SMC from the realm of mere theoretical interest to the verge of practicality; as an indicator of this fundamental change, DARPA is spending \$60 million to support the transition of technologies such as SMC to practice [17].

SMC compilers have incorporated a number of novel elements to achieve the above advances but fail in two critical areas. Specifically, as we will demonstrate, these compilers are often unstable and, when they do work, they can produce outputs that generate incorrect results. This is problematic as the integrity of the results computed by these systems may be questionable, reducing their usefulness. It might also be possible for incorrect results to be verified as correct by the SMC protocol under such circumstances. As SMC becomes ready for operational deployment, it is critical that compilers can be demonstrated to be not only efficient but also validated through principled design and testing.

In this paper, we present *Frigate*, an SMC compiler developed using design and testing methods from the compiler community. We name our compiler after the naval vessel, known for its speed and adaptability for varying missions. Our compiler is designed to be validated through extensive testing of all facets of its operation in line with how modern production compilers are validated. Frigate is modular and extensible to support a variety of research applications, and faster than the state of the art circuit compilers in the community. In addition, the frigate's use as an escort ship parallels the potential for our compiler to facilitate continued secure computation research. Our contributions follow:

- **Demonstrate systemic problems in the most popular SMC compilers:** We apply differential testing on six popular and available SMC compilers, and demonstrate a range of stability and output correctness problems in each of them.
- **Design and implement Frigate:** Our primary goal in creating Frigate is correctness, which we attempt to achieve through the use of principled and simple design, careful type checking and comprehensive validation testing. We use lessons learned from our study to develop principles for others to follow.
- **Design Frigate to be extensible:** Our secondary goal

was to provide a compiler that can be extended to provide useful and innovative functionality. After we completed the compiler we added signed and unsigned types, typed constants, and three special operators.

- **Dramatically improve compiler and interpreter performance:** The result of our efforts is not simply correctness; rather, because of our simple design, we demonstrate markedly reduced compilation time (by as much as 447x compared with previous circuit compilers), interpretation time (by over 786x), and execution time (up to 21x) when compared to currently available systems. As such, our results demonstrate that principled design can create correct SMC compilers while still allowing high performance.

Although these SMC compilers may be considered by some to be "research code," they are being used extensively within the community and by others as the basis for developing secure applications and improved primitives. The corpus of compilers we test represent a large gamut, including the most recently published solutions. In all cases, we find issues with correctness or efficiency. The implications are considerable, as the unreliable nature of many of these compilers makes testing new techniques extremely difficult. It is imperative that the community learns from these failures in design and implementation for the field to further advance. A reliable compiler is critical to this goal.

The remainder of the paper is organized as follows: Section 2 provides a background in SMC; Section 3 introduces techniques used to validate correctness; Section 4 describes our analysis of existing compilers; Section 5 defines principles for compiler design; Section 6 presents the design of Frigate; Section 7 presents our performance tests comparing Frigate to five widely-used SMC compilers; Section 8 discusses related work; Section 9 concludes.

2. Background

Since SMC was originally conceived, a variety of different techniques have been developed. Recent work has demonstrated that each technique can outperform the others in different setups (*e.g.*, number of participants, available network connection, type of function being evaluated) [23], [41], [6], [46]. In this work, we focus specifically on the garbled circuit construction developed by Yao [55]. This protocol has been shown to perform optimally for two-party computation of functions that can be efficiently represented as Boolean circuits. While our experimental analysis examines the performance of the compiler in the context of garbled circuits, it is critical to note that this compiler can be used with *any* SMC technique that represents functions as Boolean circuits.

2.1. Garbled circuits

The garbled circuit construction provides an interactive protocol for obliviously evaluating a function represented as a Boolean circuit. It involves at least two parties: the first party, the *generator*, is responsible for garbling the circuit to be evaluated such that the input, output, and intermediate wire values are obscured. The second party, the *evaluator*, is responsible for obliviously evaluating the garbled circuit with garbled input values provided by both parties.

For each wire i in the garbled circuit, the generator selects random encryption keys k_i^0, k_i^1 to represent the bit values "0" and "1" for each wire in the circuit. Given these garbled wire labels, each gate in the circuit is represented as a truth table (while each gate may have an arbitrary number of input wires, we assume each gate has two inputs without loss of generality). For a gate executing the functionality $\star$ with input wires i and j and output wire k, the generator encrypts each entry in the truth table as $Enc((k_i^{b_i}, k_j^{b_j}), k_k^{b_i \star b_j})$ where b_i and b_j are the logical bit values of wires i and j. After permuting the entries in each truth table, the generator sends the garbled circuit, along with the input wire labels corresponding to his input, to the evaluator. Given this garbled representation, the evaluator can iteratively decrypt the output wire label for each gate. Once the evaluator possesses wire labels for each output wire, the generator can reveal the actual bit value mapped to the output wire labels received.

To initiate evaluation, the evaluator must hold garbled representations of both parties' input values. However, since the evaluator does not know the mapping between real bit values and garbled wire labels, an oblivious transfer protocol is required to allow the evaluator to garble her own input without revealing it to the generator. Essentially, for each bit in the evaluator's input, both parties execute a protocol that guarantees the evaluator will only learn one wire label for each of her input bits, while the generator will not learn which wire label the evaluator selected.

This protocol guarantees privacy of both parties' inputs and correctness of the output in the semi-honest adversary model, which assumes that both parties will follow the protocol as specified, and will only try to learn additional information through passive observation. When adversaries can perform arbitrary malicious actions, a number of additional checks must be added to ensure that neither party can break the security of the protocol. These checks are designed specifically to prevent tampering with the evaluated function, providing incorrect or inconsistent inputs, or corrupting the values output by the garbled circuit protocol.

2.2. Circuit Compilers

Execution systems for garbled-circuit secure computation require functions represented as Boolean circuits. Due to this requirement, there have been several compilers created to generate circuit representations of common functions used to test this type of computation. These compilers take higher-level languages as input and transform them into a circuit representation. Writing the circuit files without using a compiler is tedious, inefficient, and will most likely result in incorrect circuits as they can have billions of gates.

3. Compiler Correctness

One of our main motivations for developing a principled compiler was the varying and unstable state of the existing research compiler space. Garbled circuit research has made significant advances in the past several years, which is largely due to a set of circuit compilers that have been commonly used to generate test applications for a significant number of protocols. Given our years of experience, we know the reliability of these results is suspect in many cases due to common errors we have found in these compilers. To facilitate continued advances in this research space, a foundational compiler with reliable performance is a critical tool. Without it, researchers will be forced to either use existing compilers, which we show are unreliable, or develop their own compilers, which is time-consuming and slows research progress. To demonstrate the need for a new and correct compiler that is openly available for the community, we examined correctness issues with the most common compilers used in garbled circuit research.

We define the *correctness* of a complier implementation using two criteria: (1) any valid program in the language can be successfully compiled, and (2) the compiler creates the correct output program based on the input file. There are two methods used to demonstrate compiler correctness: formal methods for validation and verification, and validation by testing.

3.1. Formal Verification

The concept of a verifying compiler was identified as a grand challenge by Tony Hoare in 2003 [21] due to the significant complexity in design and implementation. Since that time, the primary example of a formally verified compiler has been CompCert [32]. The development and rigorous proof of each formalized component of the compiler was an immense undertaking. However, despite the amount of time and formal verification that went into CompCert, it was demonstrated that the formal verification used in CompCert was only able to ensure correctness in select components of the compiler. When tested with Csmith [54], there were still errors found that demonstrated the limitations of formal verification. In addition, formal verification of compiler transformations and optimizations is still very much an open research area [35], [39]. Techniques such as *translation validation* [44], [40], [50] focus on the formal validation of a compiler's correctness through the use of static analysis techniques to ensure that two programs have the same semantics, and are designed to attempt to deal with the reality of legacy compilers. They have their limitations as well, particularly within the context of secure multi-party computation compilers that have not adopted any particular standard for intermediate representations. As a result, the semantic model must be adapted for every compiler implementation, and any changes in the compiler require changes to the model.

Based on these limitations and the impracticality of applying formal verification, we instead apply validation techniques that are the standard method for ensuring the correctness of compilers.

3.2. Validation By Testing

Validation by testing demonstrates that a compiler is correct through extensive unit testing. *This is by far the most common technique used in practice to ensure compiler correctness.* While testing for correctness can miss some errors in compiling specific cases, it provides a practical level of assurance that is sufficient for the vast majority of applications. Validation tests are designed by examining how to rigorously test the largest possible number of programs a compiler can generate.

There are many existing validation tests [18], [52], [51] and test suites [4], [1]. The validation tests used by ARM [1] and SuperTest [4] provide a description of the procedures they use to validate the vast majority of possible program cases. However, these suites are language-specific, often developed to find errors in popular tools such as *gcc* and *LLVM*. To date, there have not been existing validation tools designed to examine secure computation compilers. As a result, we developed our own set of validation tests based on the techniques used by these tools. Our tests, like the test suites of ARM and SuperTest, explore the possible statements and effects of those statements.

In our case, hand written tests are preferred over automatically generated tests due to us being able to examine the compiler source directly. This allows us to examine possible code paths and be more systematic with our tests than a random fuzzer. In addition, because there are no SMC compiler standards, a different fuzz generator would have to be created for each compiler input language.

Our tests follow the concept of testing the state space of the compiler starting with broad examination of operators and expressions, then refining the tests to consider common special cases. Our tests proceed through five phases:

1) Attempt all possible grammar (syntax) rules and print out the results. This shows that the compiler reads in programs correctly and demonstrates the internal program state is correct.
2) Beginning from the simplest operation to validate correctness (*i.e.*, outputting a constant) test each operator in the language and each control structure to ensure it outputs the correct result.
   a) Test the different possible primitive types and declarations.
   b) Test each operator as to whether it creates the correct output circuit.
   c) Test each control structure by itself.
   d) Test function calls, parameters, and return statements. Verify that parameters can be used inside of their functions and that return statements work correctly. Also perform tests for where different types are used as input parameters and return values.
3) Validate all the different *paths* for how data can be input into operations. Demonstrate that different control

structures work correctly together. Or, as put by SuperTest [4], "Systematically exploring combinations of operators, types, storage classes and constant values."

   a) Test if the operator deals correctly with the possible types of data that can be input as an operand.
   b) Test different types of control structures nested within each other.
   c) Test each operator under *if* conditionals with emphasis on operators that change variable values such as assignment (=), increment (++), and decrement (--).

4) Test edge cases in programs.
   a) Verify that empty functions do not crash on definition or call.
   b) Test array access and how arrays (and like operators) deal with edge cases, *i.e.*, out of bounds, minimum, and maximum values.
   c) Ensure known weaknesses in past compilers are tested to determine whether these vulnerabilities appear.
5) Perform testing to verify each previously found error was not re-added to the final implementation.

At the conclusion of these tests we have tested (1) the correctness of each mini-circuit an operator uses, (2) the ways data can come into each operator, (3) the base and nested rule for each construct (*if* statements, *for* loops, array declarations), (4) various edge cases. This set of tests checks the operators, control flow structures, types, and covers most, if not almost all, of the uses of the operators, control flow structures, and types.

4. Survey of Existing Compilers

Using the test procedures described in the previous section, we set out to quantify the common problems in existing secure computation compilers. With each compiler, we found problems that would prevent the compiler from working correctly or corrupt test applications.

While the compilers we compare ourselves to are research artifacts, such systems are widely used by the community to test and validate algorithms. Having bugs and unpredictable behavior stunts the advancement of the field.

Upon acceptance, we informed the authors of each of these compilers about the issues we found; we found that many errors, as noted for specific compilers, were corrected between the submission and acceptance of this paper.

While we do outperform many systems, the thrust of our paper is *not* performance. We compare ourselves to a number of well-known and widely tested systems and show that we are less prone to errors and produce good results. We ensured that our system was similar to current programming languages[1] and thus easy for new developers to use, and added descriptive error messages. All of this was designed to improve the way the community wrote SMC code and reduce complexity.

1. We explain why we do not use a current programming language in Section 6.1.

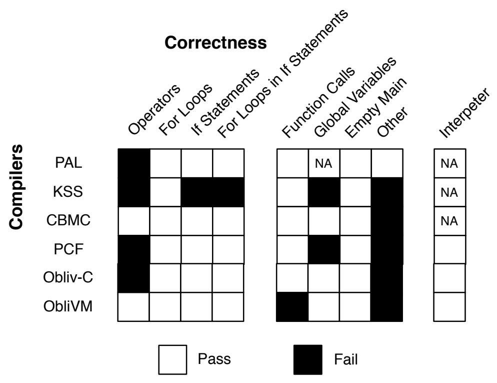

Figure 1: Summary of the correctness results.

4.1. Comparison Compiler Information

PAL: We selected PAL [38] as it was the first compiler designed for low-memory devices using an efficient intermediate representation. It also takes in Fairplay's [36] SFDL, a custom hardware description, and outputs SHDL, a gate list. It is dramatically more memory efficient than Fairplay and is able to compile much larger programs, but lacks optimizations used in recent work.

KSS: We examined the compiler from Kreuter *et al.* [31], hereon referred to as KSS. This compiler takes a hardware specific language as input and outputs a gate list in binary format. We chose KSS since it forms the basis for multiple recently-published works.

CBMC: The CBMC-GC compiler [22] (hereon CBMC) used a bounded model checker to compile a circuit program. This compiler takes a C file as input and outputs a condensed gate list (in ASCII). Because CBMC can compile programs written in ANSI C, it is commonly used in other garbled circuit research.

PCF: The PCF compiler [30], was created in order to have a condensed output format while being efficient. It takes in LCC bytecode as an input language and transforms it into a PCF file (ASCII). This file describes a circuit in a condensed format; a circuit interpreter is used to get each gate in turn. We selected PCF since it has been used to generated some of the largest circuits.

Obliv-C: Obliv-C [56] provides an extension that allows users to compile Obliv-C programs using a C compiler. These programs are then run in the normal C program fashion, *i.e.*, the default output is `a.out`. This has the advantage of being able to use the $-O3$ optimization flag.

ObliVM: ObliVM [53] compiles the input language into a Java file and then compiles the Java file using the `javac` Java compiler. The class file is then run using Java. By using Java, ObliVM is able to take advantage of the Java virtual machine optimizer.

4.2. Analyzing Compiler Correctness

We performed an analysis of the compilers to determine whether they work as expected, which we summarize in

Figure 1. We do not attempt to find edge case errors that may only affect a minutiae of programs. Instead, we focus on testing the main operations and control structures in the input language that most users would perform.

We separate our analysis of previous compilers into two areas: errors and inefficiencies. We only note an error if the original program was valid; if the compiler crashes due to an incorrect program we do not consider it to be a compiler error. However, we found that most of the compilers lacked helpful error messages when an invalid program was provided as input.

4.2.1. PAL. Errors: PAL encounters problems when *struct*s are used. This prevents the use of complex data-types unless each data item is independently defined. This appears to be an issue with the compiler's front-end, and is indicative of insufficient validation for that function within the language.
Inefficiencies and Limitations: PAL circuits are compiled into ASCII, which results in much larger file sizes than using a binary format. PAL also has some problem-size limitations, and fails to compile very large programs. While the templating concept proposed in this work is useful, it is not useful for compiling circuits of practical size.

PAL does not use the most efficient adder sub-circuit, meaning it requires $3n$ non-XOR gates instead of n (XOR gates can be executed for "free" [29]), and uses 3 input gates for MUXs. It also does not provide a complex optimization phase, so the output is not very optimized.

4.2.2. KSS. Errors: The KSS compiler has a number of areas where it does not function properly. Nested *if* statements consistently cause errors in the output circuits. Further, *for* loops used within *if* statements also cause the compiler to fail with regularity. Programs requiring multiple conditional statements, such as Dijkstra's algorithm, must be re-written to use single nested *if* statements. This is furthered hampered by the lack of conjunction operators. These shortcomings limit expressivity and the ability to write certain programs.

We discovered that when a variable is used inside of a function and also outside (*i.e.*, a global variable defined later in the code), it can lead to incorrect output circuits. This error can occur when a function and the body of the program might both use the same name for input. One such case occurred when we used the variable *a* both in the program body and in a function. Finally, we found a set of cases that the generated circuit was incorrect due to what may be an optimization error. This set of cases each used some of the same functionality.
Inefficiencies and Limitations: Rather than reduce the output size using an intermediate representation, KSS outputs the entire circuit. It also uses a very large amount of hardware-specific code, which makes porting a extremely difficult task. While this hardware-specific code provides some efficiency gain on specific platforms, it makes the task of extending the code very complex.

Like PAL, KSS's output does not take advantage of the most optimized adder sub-circuit. As a result, many circuits are about 3x larger (non-XOR gates) than necessary.

4.2.3. CBMC. Errors: Output variables can cause compilation errors if used more than once, or if read inside the program. This is a common occurrence when a function returns different values depending upon conditional statements, *e.g.*, a piecewise function. These errors can be avoided by careful programming. There is an error where input cannot be assigned directly to output variables. We also found another error that may be related to the input-to-output error though its exact cause is a mystery. CBMC provides an error message if an input variable is written within a program.

CBMC sometimes fails to compile using arrays as input. Without being able to rely on input arrays, the programmer must enter integers in an unstructured manner, making operations such as matrix multiplication more difficult.
Inefficiencies and Limitations: CBMC outputs the entire circuit in a ASCII format, which, while condensed compared to PAL, is still much larger than using binary. The format also doesn't map output variables to pins, making developing and debugging the interpreter prone to error.

4.2.4. PCF. *This section tests the published version of PCF, PCF1. There is a new version under development that, at the time of our testing, was not as efficient as PCF1.*
Errors: PCF allows global variables, but if global values are initialized during declaration, it can crash, *i.e.*, the assignment must happen later on in the program. We discovered this error trying to get AES to work correctly, though this problem affects any program that uses global variables. When an array is addressed with an out of bounds index, PCF effectively returned random results (whatever was in memory) instead of producing an error message for each test we made. This is an extremely dangerous behavior, as it can lead to hidden and hard-to-detect errors.

By default, the PCF compiler does not update the program labels that keep track of the number input wires, meaning, by default, the amount of input will not be correct if these labels are used by an execution system. Furthermore, the programmer must calculate the input sizes (in bits) of each party in every program. The *translate* script provided with PCF, which is used to convert LCC bytecode to PCF, can fail on valid input files. In addition, PCF has input buffer overflow problems as inputs above 2^{14} bits overflow the input-buffers for the two parties. This means that the circuit will most likely fail upon execution when more than 2^{14} bits of input are requested in a program, like the millionaires problem with 65,536 bits as input. These input size bounds are currently hard-coded into the PCF compiler, not defined by the program being compiled, and must be edited manually in cases where larger inputs are needed.
Inefficiencies and Limitations: While PCF produces very small output circuits, the interpreter used to parse these circuits is extremely inefficient. Our tests demonstrated that the interpreter can require as many as ten operations to read in a single gate. This overhead is magnified by the fact that each gate is calculated by the interpreter for *every* circuit that is garbled. For malicious secure execution systems where many copies of the same circuit must be garbled, it is far

```
obliv int a=0;
obliv if(input1>0)
{
  for(int i=0;i<3;i++)
    a = a + 1;
}
```

Figure 2: Example code from Obliv-C that does not optimize as much as it could.

more efficient to parse the gate once, then garble the same functionality as many times as are required for protocol security. PCF also produces spurious gates, which add to the circuit complexity and should be removed. As with many other compilers studied, PCF uses ASCII output format, increasing storage size.

As PCF uses C as input, it also does not allow for arbitrary width types. In other words, it does not support, for instance, native multiplication of two 256-bit numbers.

4.2.5. Obliv-C. *Most of the following errors were fixed between submission and acceptance of this paper*

Errors: The statement `q = q & 0` throws a compiler error; multiplying a variable by 0 also causes an error. This type of multiplication is useful for eigenvectors. These operations work successfully if non-zero values are used.

Arrays going out of bounds (both access and modification) often produce no error messages or warnings, even those that should have been discovered at compile time. Hugely incorrect accesses (*e.g.*, out of bounds by a few hundred) can produce an error and crash, but smaller errors are often not detected. Such errors can affect the gate-count and modify the output in unexpected ways.

Further, the system cannot handle large arrays – the execution system crashed when we created an `unsigned int` array of size 32,000 for testing.

Inefficiencies and Limitations: Obliv-C does not always optimize circuits even when it is easily possible to do so. For example, `q = q & q` requires n gates, where n is the bitlength of the operation. In the segment of code seen in Figure 2, Obliv-C requires about 156 non-XOR gates; our compiler requires about 43 non-XOR gates for the same. It appears Obliv-C does not always keep track of optimizations for wire states at the gate level. These kinds of statements appear in programs such as the edit distance of two strings. The authors suggest having the developer keep track of the possible range of an integer manually. However, this is not part of their compiler.

There are no arbitrary width types, which reduces the expressivity and increases gate counts of many programs. Trying to use a smaller type than an `int`, such as `char` produces seemingly strange gate counts, *i.e.,* the multiplication of two 8-bit chars and stored into an 8-bit `char` appears to not be an 8-bit multiplication and gives gate counts of either (1) a 32-bit multiplication or (2) a multiplication of unknown size (between 32 bits and 8 bits in length). We printed out the `CHAR_BIT` variable (the number of bits in a `char`) and used *sizeof* to verify the `char` is actually supposed to be 8 bits in length.

```
int@n a=0;
if(input1$0$==1)
{
  a=a+1;
}
```

Figure 3: Example code: @n dictates the length of the variable, \$0\$ picks the 0^{th} wire

4.2.6. ObliVM. *Most of the following errors were fixed between submission and acceptance*

Errors: ObliVM provides a disclaimer on their code repository about the correctness of their system that it is expected to contain a variety of errors. In our attempt to get their code working, we encountered a problem with their test script to run their code. Through conversations with the manager of the code, this problem was resolved.

The typechecking is somewhat loose, *i.e.*, it allows many different lengths of variables to be used in an expression, *i.e.*, `int2 a; int4 b; int8 c; c = b + a`. The operator length appears to depend on the size of the output variable and not the size of the operands; this can lead to incorrect results if great care if not taken (*i.e.*, `int16 t = 4096; int8 q = t % 9;` results with q as 0 when 4096 mod 9 should be 1.). Safer type checking would eliminate this possible problem. We noticed this when we tried to write a modular exponentiation program. Single bit variables often throw errors when used; for instance, they cannot be combined with multi-bit variables (it throws a Java error).

When we tried to return (output) a result of size 2, but passed in a value larger than 2 bits, we received a result larger than would fit in 2 bits (*i.e.*, it appears the return size may be ignored for the output).

The use of constants can sometimes be a problem. `x = 100+x;` throws an error, but `x = x+100;` compiles successfully.

When returning the result of an expression (*e.g.*, `x + y`), storing the value in a variable and then returning it (where the variable is of the return size) may produce a different value than returning the expression directly; both should produce the exact same result.

Inefficiencies and Limitations: ObliVM, like Obliv-C, does not appear to provide a large amount of gate-level optimization. The statements seen in Figure 3 require approximately $2n$ non-XOR gates (where n is the length of the variable). However, gate optimizations should prevent any non-XOR gates from being required in this segment of code. These kinds of statements appear in programs such as the edit distance of two strings. Likewise, a statement like `a = a & a` should require no gates of any kind, but it requires n AND gates in ObliVM.

Selecting more bits in a variable than exists allows compilation to succeed, but throws an error at runtime.

```
error: incompatible types: t__T[] cannot be converted to
     int
->  int __tmp12 = f_tmp_6;}
```

Figure 4: Example error message from ObliVM.

The error messages are not always helpful; they are mostly Java errors from the generated Java program. An example error can be seen in Figure 4.

4.3. Summary

PAL, KSS, CMBC, Obliv-C, ObliVM, and PCF crashed on programs that should correctly compile. KSS, ObliVM, and PCF generated incorrect circuits. These are important problems. Consider how easy it is for an array to go out of bounds or the number of programs that benefit from nested conditional statements. Or, if the expressivity is severely limited by incorrect operators then programs cannot be written as efficiently as they could otherwise. Principally, if the program files used in an SMC computation are not correct then the resulting SMC computation will not be correct either.

5. Compiler Development Principles

Given the problematic state of secure computation compilers in the research community, we set the primary goal of our work to be the development of structured design practices for secure computation compilers, and to demonstrate the effectiveness of these practices with a new compiler implementation. By examining practices used by the compiler community and combining those best practices with the observed failings of previous secure computation compilers, we have assembled a set of four principles to guide the development of our compiler, Frigate. Through this implementation, we demonstrate that these principles should be considered standard practice when developing new compilers for secure computation applications.

1) *Use standard compiler practices:* Use standard methodology from compilers (lexing, parsing, semantic analysis, and code generation). Use data structures that are described throughout compiler literature (*e.g.*, an abstract syntax tree) [5]. Applying these standard, well-studied constructs allows for straightforward modular treatment of the compiler components when extending the functionality. Furthermore, it allows for application of standard compiler debugging practices.
2) *Validate the compiler output:* All production compilers rely on proper program validation to ensure that the compiler functions correctly. A variety of validation test sets have been developed in both the research community and in industry that can be applied to newly-developed compilers [4], [1], [43].
3) *Handle errors well with helpful error messages:* Many sources describing good compiler practices emphasize the need to produce error messages, also known as negative results (e.g,. [5], [4]). While allowing the compiler to crash silently on an incorrect program does not affect its overall correctness, it severely hampers usefulness.
4) *Simplify the design:* A standard software engineering principle is to avoid erroneous code by using simple designs. This allows for more intuitive debugging when errors do occur, as well as facilitating the addition of future functionality.

6. The Frigate Compiler

To demonstrate the practical effectiveness of our compiler design principles, we designed the Frigate compiler and secure computation language. We also created a fast interpreter to read Frigate's output files efficiently. Our work demonstrates three additional contributions to the state of secure computation compiler research: (1) a new and simplified C-style language with specifically designed constructs and operators for producing efficient Boolean circuit representations; (2) a compiler that produces circuits with orders of magnitude less execution time than previous compilers; and (3) a novel circuit output format that provides an efficient balance between compact representation and speed of interpretation.

6.1. Input Language

Frigate's novel input language incorporates the best of what we have seen and used in the community and partially because of this, we can achieve substantial non-XOR gate efficiency. Our novel output format provides a balance between file size and removing extra instructions necessary in some formats, *e.g.*, PCF. Frigate is meant to be a well-tested, user-friendly tool, which incorporates well known circuit optimizations and provides good performance, allowing researchers to easily create their own special optimizations without having to write their own compilers.

To better facilitate the development of programs that can be efficiently compiled into Boolean circuits, we developed a custom C-style language to represent secure computation programs. The language allows for efficiently defining arbitrary bit-length variables that translate readily into wire representation, and restricts operations in a manner that allows for full program functionality without excessive complexity. We do not use C or a common intermediate representation like LLVM's bytecode as input, to allow for innovative operators and non-standard bit-width operations.

This minimal set of operations adheres to our fourth design principle of maintaining simplicity to ensure for easier validation. Our language has control structures for functions, compound statements, *for* loops, and *if/else* statements. We include the ability to define types of arbitrary length and combination as in SFDL, the language used by Fairplay, combined with an operator that selects some bits from a variable used in the KSS compiler input language. We allow signed *int_t*, unsigned *uint_t*, and struct *struct_t* types in our input language (we can handle arrays inside

Operators	Description
+ - * / %	arithmetic operators
** // %%	extending and reducing operators
| ^ & ~	bitwise operators
=	assignment operator
++ −−	increment and decrement operators
== !=	equality test operators
> < <= >=	conditional operators
<< >>	shift operators
<<>	rotate left operator
.	struct operator
[]	array operator
{} {:}	wire operators

TABLE 1: A table showing the operators in Frigate's input language. As data types are either signed and unsigned, the arithmetic and conditional operations behave differently depending on whether the operands are signed or unsigned. In the case signed and unsigned types are used in the same operator, the compiler uses the unsigned operator (a warning is also issued by the compiler). Extending and reducing operators are discussed in Appendix B.

structs). For modularity, we have *#include* statements to allow the use of external files and *#define* to replace a term with an expression. The list of operators in our language is in Table 1, with an example of our input language in Appendix A.

Every program begins with a declaration of the number of parties participating in the computation. Since not every participant is required to provide input or receive output, the input and output types for any subset of the participants may then be specified.

To further maintain simplicity, only three primitive types are defined in our programming language. *int_t* types are signed numbers defined to a specific bit length, *uint_t* types are unsigned numbers defined to a specific bit length, and *struct_t* types may consist of *uint_t* , *int_t*, and *struct_t* types. Developers may specify their own types using these three types and the *typedef* command. These three types can be combined to create any complex data type. To formally define the typing of each operator in our language, we give a selection of typing rules in Figure 5. The remainder of these rules are available in the tech report.

One feature we were compelled to omit from our language was global variables. We removed this feature after we realized the significant overhead they represent within a Boolean circuit program. Allowing global variables requires keeping track of whether each function is called under an *if* statement and adding a MUX gate every time a global variable wire is assigned a value. Our language is capable of expressing equally functional programs by passing in "global" variables and returning any new values for these variables.

6.2. Compiler Design

With our input language defined, we next examine the design of the Frigate compiler itself. Written in approximately 25,000 lines of C++, the compiler is designed to be simple enough to validate each output code path and modular for expansion to fit specialized secure computation applications. While there are other languages (with stronger typing, for instance), which would have made it easier to show the correctness of the compiler, we use C++ for speed and available libraries. Note that Frigate can handle a variety of security models since we can attach any SMC implementation to the compiler without affecting the adversarial model.

$$\textsc{Add}\quad \frac{\Gamma \vdash t_i : Num_{L_i}}{\Gamma \vdash t_1 + t_2 : Num_{L_i}} \qquad \textsc{Less}\quad \frac{\Gamma \vdash t_i : Num_{L_i}}{\Gamma \vdash t_1 < t_2 : Num_1} \qquad \textsc{Assn}\quad \frac{\Gamma \vdash t_i : T}{\Gamma \vdash t_1 = t_2 : T}$$

$$\textsc{If-Else}\quad \frac{\Gamma \vdash t_i : T \quad \sigma : Num_1}{\Gamma \vdash \texttt{ if } (\sigma)\{t_1\} \texttt{ else } \{t_2\} : T} \qquad \textsc{Func-Call}\quad \frac{\Gamma \vdash t_i : T_i \quad f : F}{\Gamma \vdash f(t_0 ... t_{n-1}) : R}$$

Figure 5: Example typing rules for basic operators and control flow statements

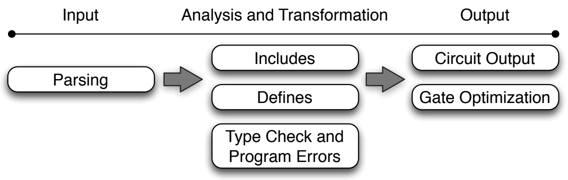

Figure 6: Overall design of the Frigate compiler. There are six separate blocks of the compiler separated blocks into three different stages instead of the traditional two stages.

6.2.1. Compilation stages. Frigate represents programs in the standard compiler data structure, the *abstract syntax tree (AST)*. In accordance with our first design principle, this allows for straightforward static analysis and transformation of each program. Each type of operation has its own node where construction, type checking, and output of its subcircuit (among other functions) takes place.

Compilation of a program follows three phases as shown in Figure 6. The input section of Frigate takes in a program and creates an AST representation of the program. We used Flex [2] and Bison [3] to generate the scanner and parser used in this phase. In the second phase, any *#include* statements are replaced with the included file's generated AST. All *#define* statements replace any terms in the AST with a deep copy of the defined expression tree. To conclude this phase, the type checker takes the AST and checks that it is a valid program as defined by Frigate's input language. The final phase of compilation takes in the AST and outputs the circuit while performing gate-level optimizations. If a developer wishes to extend the functionality of Frigate, this modular phase design allows for additional stages to be inserted in between the existing stages.

6.2.2. Type Checking and Error Output. To satisfy our third design principle, we created our type checker to output detailed error messages to indicate the location and type of error generated by an incorrect program (*e.g.*, *./tests/add.wir, Error line:11 Type "mytype" is used but not defined*). To ensure developers do not include unstable functionality in their programs, Frigate enforces strict type checking that prevents different types from interacting unless those types are different signed or unsigned integer types of the same length. A warning is issued in this case.

6.3. Circuit Representation

Previous work in compiler development has demonstrated that it is possible to have either a large yet simple circuit representation that is efficient to parse, or a highly compact circuit representation that incurs significant cost when interpreted by the evaluator. To strike a balance between these two extremes, we developed a novel circuit representation that is significantly smaller than the simplified circuit representations while still being efficiently parseable. Our output format represents circuits using four elements: a set of input and output calls, gate instructions, function calls, and copy instructions. Our representation of functions, as different files, allows us to shrink the output size without the need for a costly circuit interpreter (more details in Section 6.3.2).

To further improve the efficiency of evaluating Frigate circuits, we designed the compiler to favor XOR gates, as they can be evaluated with fewer operations and do not consume bandwidth when certain garbled circuit protocol optimizations are used [29]. We use the four-XOR, one AND-full adder introduced by Boyar et al. [10].

6.3.1. Output Components. Here we present the details of our circuit representation.

Wires: Each variable is composed of many wires that are allocated as needed with a set address. Each wire exists in either a *used* wire bin or a *free* wire bin. Once a *used* wire is freed it is placed in the *free* bin. Order, as defined by the address of a wire, is not preserved in the *free* wire bin. Our compiler will free the wires it can after each operation.

We group wires together by the number requested for a specific variable. This allows for a massive decrease in the amount of time required for checking whether or not wires can be placed in the free wire bin, *i.e.*, instead of requiring 100,000 checks for a variable with 100,000 bits (wires) in length, only a single check is needed.

Wires can exist in one of six states. *ZERO* and *ONE* represent a wire's state as 0 or 1. The *UNKNOWN* state represents wires that depend on input values such that their value cannot be computed at compile time. *UNKNOWN_INVERT* represents an unknown wire but at some point was inverted. *UNKNOWN_OTHER* and *UNKNOWN_INVERT_OTHER* are wires whose values are pointers to another wire value or the inversion of another wire value. By keeping track of inverted states, we can optimize away inverts in some cases.

Gate Output: Given two input wires and a truth table, the *outputGate* function will output a gate and update the state of the output wire. An additional function is called to determine whether the gate is needed or whether it can be optimized out. If the gate cannot be optimized out then the truth table will be adjusted for whether either of the input wires' states are inverted. Finally, the gate will be added to the output.

Function Parameters and Return States: Since we output the gate representation of each function independently only once, uncorrelated with a single function call, we cannot take advantage of knowing the state of a wire as it is passed into a function. Therefore, function parameter states are marked as *UNKNOWN*. It is possible to pass wires with "0" and "1" states, but it is not as efficient as the optimizer cannot use the information that they are "0" and "1" since they must be marked as *UNKNOWN*. This inefficiency is necessary since we only output each function a single time preventing us from taking advantage of specific parameter states. We could solve this by outputting multiple function files with different wire parameters, but this would expand our circuit representation.

6.3.2. Circuit Interpreter. Using our circuit output format, the process of interpreting a circuit is reduced to a highly efficient task. When the interpreter is initially called, it reads an *.mfrig* file, which contains information about the number of parties, input and output sizes, and the number of functions. It then opens the *.ffrig* function files. After the interpreter is initialized, it is ready for the first *getNextGate* command. Each time *getNextGate* is called, the compiler reads and executes the next instruction, and returns the appropriate gate to the execution environment.

Each function occupies a specific set of wire values such that no function's wires will overlap. This enables a "stack" of function calls without the need for the push and pop operations that would be required if our functions used overlapped wire addresses. This does not affect the output circuit size. The interpreter keeps a call stack of active functions in its internal state. Each function, rather than being held completely in memory, is stored as a pointer to the active instruction. When a function is called, the stack of functions is updated, the active function is set to the called function, and the called function is set back to the first instruction.

6.4. Procedures

While our technique of dividing programs into distinct functions and then composing the circuit with calls to those functions allows for a significant reduction in the representation size of many circuits, not all programs can be easily partitioned into distinct functions. Even if a clean partitioning does exist, the function overhead for copying parameters and return values can exceed the number of commands inside the function. Large representation size is commonly encountered with loops, creating redundant data that expands the size of the circuit representation. To reduce the output file size in this case, we develop a novel construct called *procedures*.

A procedure is an area of a loop that can be moved by the compiler to a separate function so that, instead of unrolling all the instructions for every iteration of the loop, all that is required is a single function call to the procedure function. Procedure can be intermixed with other non-procedures inside of the same loop.

As the procedure circuit is exactly the same each iteration, there are limits when using variables whose values are *ONE* or *ZERO* inside of the procedure and change between iterations. Most notably, this limitation includes using the value of the loop variable.

To demonstrate the output file size reduction possible using procedures, we consider an example program that adds five 32-bit variables to an accumulator 1000 times (the full program is in Appendix A). If no procedure is used, this program requires an output file of about 13MB since each iteration of the main loop must be unrolled. However, if a procedure is used, then output is one 30 KB file (main) and one 13KB function file (the procedure), a reduction of the total disk usage by over 300x.

7. Experiments

7.1. Frigate Validation

During the creation of Frigate, we unit tested each new structure to ensure it functioned properly. Our unit tests comprised checking most, if not all, possible program paths. We manually checked each operator with sample output. Then, to demonstrate the correctness of circuits created by Frigate, we ran an extensive validation test suite consisting of over 17,000 tests and several million additional tests containing all possible combinations of input using 8-bit types for complex operators. After hundreds of iterations of development and testing and months of work, Frigate successfully passed all validation tests, and produces correct and functioning circuits in every case where previous compilers failed. For further details on the state space we examined in Frigate, see Appendix C.

7.2. Compiler Efficiency Tests

By constructing a compiler using our four development principles, we wanted to evaluate whether adhering to the principles we laid out would have an adverse effect on performance. We tested the time that is required to compile circuits in Frigate against the three compilers (CBMC, PCF, and KSS) that output a complete circuit. We also tested ObliVM and Obliv-C, but do not include them in the compile-time results as they do not directly output circuits and thus are not directly comparable. We show some of these results in Appendix A. PAL did not give competitive compilation results, so we omit them from our benchmarks. For Obliv-C and ObliVM, we measure the efficiency (gate counts) of primitive operations as the runtimes of the compilers correspond to the C and Java compilers, respectively. All of our benchmarking tests were performed on a MacBook Pro with an Intel i7 4-core 2.3Ghz with 16GB RAM, 256KB L2/core, and 6MB L3.

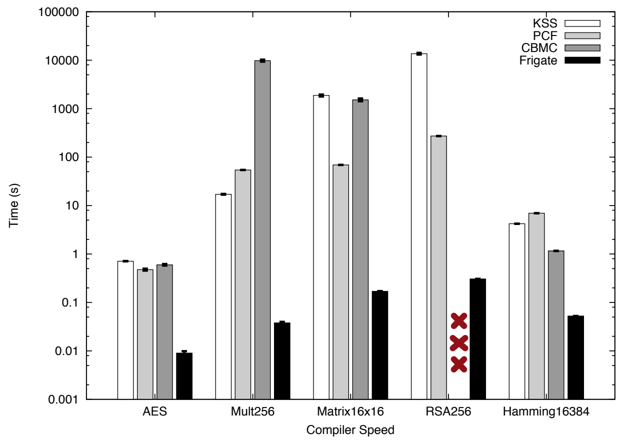

Figure 7: Comparing the different compilers we tested for compilation time. We did not succeed in compiling RSA256 with CBMC. Note the y-axis is logscale.

7.2.1. Test Programs. To evaluate performance across a wide variety of compilers, we used common test programs used by the other researchers in this space [31], [30], [48]. We used the following test programs: multiplication with matrices of X by X with 32-bit values, AES, Hamming distance of two X bit numbers, multiplication of two X-bit numbers that produces a 2*X-bit result (this is in contrast to the program of TinyGarble [49] who use X*X=X-bit multiplication), and RSA (modular exponentiation) of X bits, where the base, exponent, and modulus are all X bits in length. For each test program, we varied the input size X throughout our testing.

7.2.2. Tests. We summarize the results in Figure 7 by comparing the largest input values for each program that successfully compiled across all compilers. We evaluate the compilers with their default setup in an attempt to produce the smallest circuit (as opposed to disabling the circuit optimizers). In every case, Frigate completes compilation the fastest. In the best case, Mult 256, which computes the multiplication of two 256-bit numbers, Frigate compiles 447x faster than the next fastest compiler, KSS.

In addition to comparing speed efficiency, we also considered the non-XOR gate counts of each program compiled. Because the free-XOR optimization for garbled circuits [29] allows XOR gates to be evaluated with non-cryptographic operations and without consuming network bandwidth, we consider non-XOR gates the bottleneck in computation. Frigate greatly reduces the number of non-XOR gates for the Mult-4096 program, demonstrating a reduction for the number of non-XOR gates by about 3x. In the case of AES, and RSA-512, the improvement was only slightly better than existing compilers, reducing the gate count by up to 1.18x. We observed a increase in gate counts for Matrix Multiplication by 0.8%, Hamming Distance by 2.35x, RSA-256 by 1.19x, and Mult-256 by 1.37x. The full compilation results are in Table 4, in the Appendix.

Other than Hamming Distance and AES, our gate counts

	Frigate	TinyGarble	
ProgramName		C	Verlog
Hamming-160	719	1,264	158
Sum-1024	1,025	3,067	1,023
Compare-16384	16,386	52,224	16,384
X-to-X-bit Mult-64	4,035	-	3,925
MatrixMult5x5	128,252	-	120,125
AES	10,383	-	5,760

TABLE 2: Non-XOR gate count comparison between Frigate and TinyGarble [49] using HDL and C as inputs. "-" represents results not present in [49]. For accurate comparison, our multiplication operation in this test produces n-bit output as in [49].

are similar to the best gate counts of [49], who wrote programs in behavioral and RTL level Verilog. For the three test programs given in [49] that use a high level language (C), we are superior. Table 2 gives the exact results.

While TinyGarble produces superior gatecounts in some cases, this is achieved using Verilog, a hardware description language for electronic systems. Thus, the interpreter converts something that is already close to a hardware-level description into a circuit format, as opposed to dealing with a high level language. It is not surprising, then, that in a some cases, the Verilog version, which is closer to a handcrafted circuit, performs better than Frigate.

For Obliv-C and ObliVM, we measure the cost of some primitive operations, shown in Table 3. Frigate, Obliv-C, and ObliVM have similar gatecounts for compare and sum operations. These operations have $O(N)$ gates, where N is the bitlength of the operation. In contrast, ObliVM's multiplication and division templates are larger than that of both Frigate and Obliv-C, and Frigate's division template is larger than that of Obliv-C.

It should be noted that neither Obliv-C nor ObliVM were able to perform the a = a & a optimization to emit AND gates. We include this optimization to show that neither of these two systems perform a number of known optimizations that should have been included. This is a disadvantage of their model of compiling to an executable instead of a circuit: in order to perform these optimizations, the system will have to perform them every time the circuit is executed.

To summarize, *ObliVM and Obliv-C do not perform known gate-level optimizations*. Without these and many other optimizations implemented in Frigate, they sometimes produce comparatively inefficient output programs.

7.3. Interpreter and Execution Speed

Interpreter Time: Our next set of experiments compares the performance of the Frigate and PCF interpreters. Figure 8 shows our experimental results. The Frigate output format allows for significant reduction in interpreting time. In the worst-case, we improve over PCF by 106x, with a reduction of 786x in the best case. We used the Unix *time* function to measure the total computation time.

A lot of this speedup comes from the way Frigate and PCF are designed. (1) Frigate optimizes the circuit a single time in the compiler; PCF has to optimize at runtime. (2) Frigate's execution system can take advantage of certain compiler (*gcc*) optimizations that PCF cannot due to a design decision requiring each instruction to use function pointers (*i.e.*, lots of function calls that could be optimized out). (3) Many instructions in PCF require a malloc due to the interface to the PCF interpreter; Frigate's interpreter requires no mallocs after initialization.

Execution: The total execution time is improved by a faster interpreter. We connected the Frigate interpreter and the PCF interpreter into the same semi-honest execution system loosely based on the KSS execution implementation but further optimized and modified to use generic C++ vectors as its primitive type (instead of Intel-only intrinsic data types) to increase portability. Figure 9 shows the results.

Total execution time improves by 21x in the best case and by 1.8x at worst. We can further improve our performance by reducing the overall execution time by adding in additional optimizations like the half-gate optimization [57] or the fixed-key blockcipher optimization [7]. The speedup is the result of the amount of extraneous instructions PCF re-

	Frigate	Obliv-C	ObliVM
Sum-32	31	31	32
Compare-32 (>)	32	32	32
X-to-X-bit Mult-8	59	-	120
X-to-2X-bit Mult-8	136	-	176
X-to-X-bit Mult-32	995	993	2,016
X-to-2X-bit Mult-32	2,082	-	3,008
Div-8	61	-	172
Div-32	1,437	1,210	2,236
a = a & a	0	$O(N)$	$O(N)$

TABLE 3: Non-XOR gate count comparison of different operations for Frigate, Obliv-C, and ObliVM. For these tests we look at the non-XOR gate counts different primitive operations require (not gate counts for a specific program). For Obliv-C, we do not measure 8-bit operations (`char` variables) as they does not appear to give correct gatecounts as noted in Section 4.2.5. Using signed types.

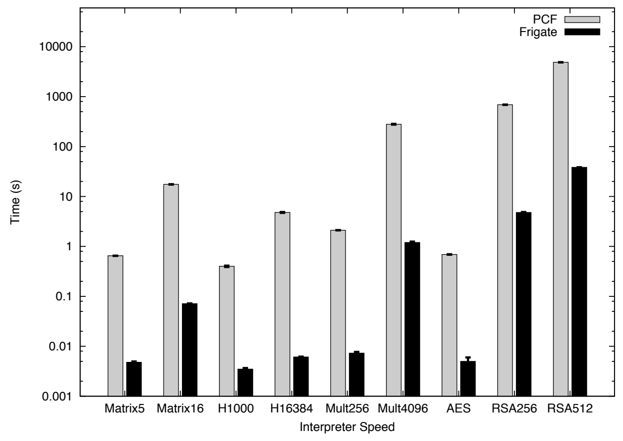

Figure 8: Interpreter time per circuit for PCF and Frigate interpreters. Note the y-axis is logscale. H stands for Hamming Distance.

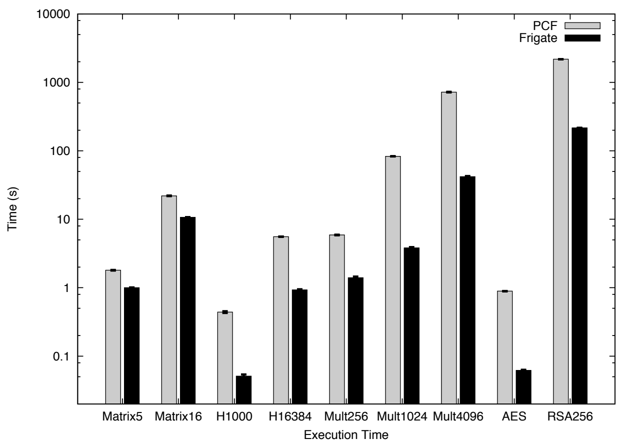

Figure 9: Execution performance for semi-honest execution system in Frigate and PCF. In these experiments we only vary the interpreter and circuit format. The execution system is the same in both cases. H stands for Hamming Distance.

quires, sometimes up to 18x instructions per gate instruction. The interface, though elegant, requires malloc on many of these. Here, our novel output format provides an advantage.

7.4. Discussion

Speed of Frigate: During the creation of Frigate, we attempted to speed up Frigate in many ways. Other than writing efficient code, our separation of functions, output representation, choice of programming language, efficient data structures, lack of a global optimization phase, efficient use of circuit templates, and our use of *procedures* all contribute to why our compiler performed better.

Not Comparing to Other Interpreters: Both Obliv-C and ObliVM compile to executables and not directly to circuits. For PCF and Frigate, we could easily swap the interpreter while maintaining the same SMC execution backend.

Extensibility: After the initial creation and implementation of Frigate, we made additional changes that show extensibility. We enabled constants to be defined with a specific sign and specific bit-length, which was not in our original specification, and added three additional operators: extending multiplication, reducing division, and reducing modulus. These operators are discussed in detail in Appendix B.

For developers to extend Frigate with their own functionality, they simply create or modify an AST node and the parsing rules, modify typing for new or existing operators, and then define what sub-circuit the operator outputs.

Tools: To demonstrate how Frigate can be used to create useful developer tools, we created an extension to output the gate counts of program components inline in a printout. We implemented this tool to understand where the most costly operations are in a program. Our tool also maps in the cost of function calls even though they are not called during compilation. As procedures are not output during each iteration of a loop, the costs inside a procedure

```
function int mul(int x, int y)
<1047555,2092036>{
  return x * y;
}
function void main()
<270270213,540799236>{
  int t = input1;
  for(int i = 0; i < 256; i++)
  <268174080,536610048>{
    t = mul(t, input1);
  }
  <1047555,2092036>{
    t = t * input1;
  }
  output1 = input1 * input2 + t;
}
```

Figure 10: Example of Frigate's gate counts in the program at each compound statement. Key: $\langle$*non−XOR gates*, *free operations*$\rangle$

represent only a single iteration, but outside the procedure, it is counted for every iteration. Figure 10 shows an example.

Very Large Circuits: Using our unique output format, we were able to compile a program that has 2^{467} non-XOR gates in under 20 minutes (a 1024-bit addition performed 20000^{32} times). This is not the limit, but shows Frigate can create very large programs.

Full Circuit List: Since many systems we have seen and used require the full circuit as input, we created a tool (via a command-line argument to the interpreter) to output the complete circuit into an easy to understand text format.

8. Related Work

When the garbled circuit protocol was developed by Yao [55], it demonstrated that secure multiparty computation was possible. However, the protocol remained a theoretical novelty until Fairplay [36] demonstrated that the protocol could be feasibly run for small circuits. In more recent work, the garbled circuit protocol has been vastly expanded from its original capability and applied to various areas [9], [20], [16], allowing for multiple parties [8], security in the presence of covert [19], malicious [27], [33], [34], [47], [48], and other adversaries [25], as well as outsourced execution from computationally limited devices [26], [13], [12], [14] and on mobile devices directly [11].

Several optimizations have been developed to reduce the size of garbled circuits. The free-XOR technique [29], [15] allows garbled XOR gates to be evaluated with a single XOR operation and requires zero bandwidth. Optimizations such as garbled row-reduction [42] allow for the size of the transmitted AND gates to be reduced by a constant factor. Other optimizations, such as FleXOR [28], have been shown to reduce bandwidth and computation time for certain functions. The pipelining technique developed by Huang et al. [24] generates and transmits the circuit in layers, allowing large circuits to be handled in a small amount of memory. Most recently, the PartialGC system [37] allows for garbled wire values to be re-used between protocol executions. However, while these protocol optimizations allow for constant factor improvements in speed and bandwidth, they do not optimize the size of the boolean representation itself.

Fairplay [36] was the first SMC compiler. While this provided a first step towards a practical and usable means for representing arbitrary programs as circuits, it suffered from a number of correctness issues. To reduce the size of the unoptimized circuit representation, the PAL compiler [38] used pre-optimized templates instead of completely creating each circuit at runtime. The compiler by Kreuter, shelat, and Shen [31] incorporated some of circuit optimizations. The Portable Circuit Format compiler (PCF) [30] combined the concept of templating with several circuit optimizations. Another compiler, Wysteria [45], provides support for mixed-mode secure computation.

A recent work by Songhori et. al [49] shows how to use hardware tools to create SMC circuits. These produce significantly smaller output files and often significantly smaller non-XOR gate counts when writing in an HDL language.

9. Conclusion

Garbled circuit protocols have made significant advances based upon the development of a set of circuit compilers that have allowed researchers to quickly develop new test applications. However, the sometimes error-prone nature of these compilers has made building new research on them problematic. In this work, we examine the state of secure computation compilers using rigorous validation testing. From this examination, we present a set of guiding principles for secure computation compiler design and develop the Frigate compiler based on these principles. By building a principled compiler and thoroughly validating correctness, our compiler reduces compile time by as much as 447x when compared to previous circuit compilers. Furthermore, our novel circuit representation format allows for circuit interpretation time to be reduced by as much as 786x and execution time by up to 21x. These results demonstrate that a principled approach to design and validation of secure computation compilers produces tools that are both correct and efficient, and offer the community a solid foundation on which to develop further research.

The source code for Frigate is available at *https://bitbucket.org/bmood/frigaterelease*

Acknowledgments This work was supported in part by the US National Science Foundation under grants CNS 1464087, 1464088, and 1540217. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We would like the thank the authors of the various compilers we examined for their help with acquiring and using them.

References

[1] Arm Compiler Verification Process. http://www.arm.com/products/tools/software-tools/mdk-arm/compilation-tools/compiler-verification.php.

[2] flex: The Fast Lexical Analyzer. http://flex.sourceforge.net.

[3] Gnu bison. http://www.gnu.org/software/bison/.

[4] SuperTest Compiler Test and Validation Suite. http://www.ace.nl/compiler/supertest.html.

[5] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[6] I. D. ard, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology - Crypto*, 2012.

[7] Bellare, Mihir and Hoang, Viet Tung and Keelveedhi, Sriram and Rogaway, Phillip. Efficient Garbling from a Fixed-Key Blockcipher. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.

[8] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-Party Computation. In *Proceedings of the ACM conference on Computer and Communications Security (CCS'08)*, 2008.

[9] D. Bogdanov, R. Talviste, and J. Willemson. Deploying secure multi-party computation for financial data analysis. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 2012.

[10] J. Boyar, R. Peralta, and D. Pochuev. On the multiplicative complexity of boolean functions over the basis $(\wedge, \oplus, 1)$. *Theoretical Computer Science*, 235(1):43 – 57, 2000.

[11] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For Your Phone Only: Custom Protocols For Efficient Secure Function Evaluation On Mobile Devices. *Journal of Security and Communication Networks (SCN)*, 7(7):1165–1176, 2014.

[12] H. Carter, C. Lever, and P. Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[13] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of the USENIX Security Symposium (SECURITY'13)*, 2013.

[14] H. Carter, B. Mood, P. Traynor, and K. Butler. Outsourcing Secure Two-Party Computation as a Black Box. In *Proceedings of the International Conference on Cryptology and Network Security (CANS)*, 2015.

[15] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the "free-xor" technique. In *Proceedings of the international conference on Theory of Cryptography*, 2012.

[16] G. D. Crescenzo, J. Feigenbaum, D. Gupta, E. Panagos, J. Perry, and R. N. Wright. Practical and privacy-preserving policy compliance for outsourced data. In *Proceedings of the International Conference on Financial Cryptography and Data Security - Workshop on Applied Homomorphic Cryptography*. Springer, 2014.

[17] DARPA. DARPA "Brandeis" program aims to ensure online privacy through technology. http://www.darpa.mil/NewsEvents/Releases/2015/03/11.aspx, 2015.

[18] P.-L. Garoche, F. Howar, T. Kahsai, and X. Thirioux. Testing-Based Compiler Validation for Synchronous Languages. In *NASA Formal Methods*. 2014.

[19] V. Goyal, P. Mohassel, and A. Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. In *Advances in Cryptology (EUROCRYPT'08)*, 2008.

[20] D. Gupta, A. Segal, A. Panda, G. Segev, M. Schapira, J. Feigenbaum, J. Rexford, and S. Shenker. A new approach to interdomain routing based on secure multi-party computation. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 37–42. ACM, 2012.

[21] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, Jan. 2003.

[22] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure Two-party Computations in ANSI C. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[23] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS '12: Proceedings of the 19th ISOC Symposium on Network and Distributed Systems Security*, San Diego, CA, USA, Feb. 2012.

[24] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the USENIX Security Symposium (SECURITY '11)*, 2011.

[25] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[26] S. Kamara, P. Mohassel, and B. Riva. Salus: A System for Server-Aided Secure Function Evaluation. In *Proceedings of the ACM conference on Computer and Communications Security (CCS'12)*, 2012.

[27] M. S. Kiraz. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[28] V. Kolesnikov, P. Mohassel, and M. Rosulek. FleXOR: Flexible Garbling for XOR Gates That Beats Free-XOR. In *Advances in Cryptology – CRYPTO*, 2014.

[29] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'08)*, 2008.

[30] B. Kreuter, B. Mood, a. shelat, and K. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *Proceedings of the USENIX Security Symposium (SECURITY '13)*, 2013.

[31] B. Kreuter, a. shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the USENIX Security Symposium (SECURITY'12)*, 2012.

[32] X. Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7), July 2009.

[33] Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Advances in Cryptology (EUROCRYPT'07)*, 2007.

[34] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the conference on Theory of cryptography*, 2011.

[35] Lopes, Nuno P. and Monteiro, José. Weakest Precondition Synthesis for Compiler Optimizations. In *Verification, Model Checking, and Abstract Interpretation*, 2014.

[36] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–A Secure Two-Party Computation System. In *Proceedings of the USENIX Security Symposium (SECURITY'04)*, 2004.

[37] B. Mood, D. Gupta, K. Butler, and J. Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[38] B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC'12)*, 2012.

[39] K. S. Namjoshi, G. Tagliabue, and L. D. Zuck. A Witnessing Compiler: A Proof of Concept. In *Runtime Verification*. 2013.

[40] G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, May 2000.

[41] J. Perry, D. Gupta, J. Feigenbaum, and R. Wright. Systematizing secure computation for research and decision support. In M. Abdalla and R. De Prisco, editors, *Security and Cryptography for Networks*, volume 8642 of *Lecture Notes in Computer Science*, pages 380–397. Springer International Publishing, 2014.

[42] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation is Practical. In *Advances in Cryptology (ASIACRYPT'09)*, 2009.

[43] Plum Hall, Inc. The Plum Hall Validation Suite for C. http://www.plumhall.com/stec.html.

[44] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, London, UK, 1998.

[45] A. Rastogi, M. Hammer, M. Hicks, et al. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symposium on Security and Privacy (S & P)*, pages 655–670. IEEE, 2014.

[46] T. Schneider and M. Zohner. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *Financial Cryptography and Data Security*, 2013.

[47] a. shelat and C.-H. Shen. Two-Output Secure Computation with Malicious Adversaries. In *Advances in Cryptology (EUROCRYPT'11)*, 2011.

[48] a. shelat and C.-H. Shen. Fast Two-Party Secure Computation with Minimal Assumptions. In *Conference on Computer and Communications Security (CCS'13)*, 2013.

[49] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *36th IEEE Symposium on Security and Privacy (Oakland '15)*.

[50] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[51] C. Wang, S. Chandrasekaran, and B. Chapman. An OpenMP 3.1 Validation Testsuite. In *OpenMP in a Heterogeneous World*, 2012.

[52] C. Wang, R. Xu, S. Chandrasekaran, B. Chapman, and O. Hernandez. A Validation Testsuite for OpenACC 1.0. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, 2014.

[53] X. Wang, C. Liu, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy (S & P)*, 2015.

[54] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, 2011.

[55] A. C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'82)*, 1982.

[56] S. Zahur and D. Evans. Obliv-c: A language for extensible data-oblivious computation. http://oblivc.org/category/paper.html, 2015.

[57] S. Zahur, M. Rosulek, and D. Evans. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates. In *Eurocrypt*, 2015. http://eprint.iacr.org/.

Appendix A. Example Programs

Input Example: The program below gives an example of much of the syntax in Frigate's input language, using statements, types, and input and output. Each output is the addition of the two inputs.

```
#define wiresize 32
#parties 2
typedef int_t wiresize int
typedef struct_t mystruct {
    int x;
}
typedef struct_t newstruct {
    int x;
    newstruct var[5];
}
#input 1 int
#output 1 int
#input 2 int
#output 2 int
function void main(){
    output1 = input1 + input2;
    output2 = input1 + input2;
}
```

Procedure Example: This is the example program discussed in Section 6.4.

```
function void main(){
    int x = input1;
    int y = input2;
    for(int i=0;i<1000;i++) {
        x = x + y + y + y + y + y;
    }
    output1 = x;
}
```

Table 4 gives compiler performance results for our different test programs.

Appendix B. Frigate Design Details

This section lists some of our design decisions and explains why we decided to do things differently from other compilers.

No Recursion: We do not allow recursion in our execution model. Multiple copies of a specific function could be created to simulate recursion but this is not done as part of a native operation. Since the depth of recursion must be known at compile time, this does not remove functionality from the language but may reduce expressiveness.

Typed Constants: Constants can be typed to a specific length and sign. By default, constants are not typed, sized to their bit-length + 1, and use unsigned operators when all operands are untyped constants. In the presence of typed operands and untyped constants, the operator will use the sign of the typed operand. We added the # and ## operators to specify the bit-length and type of constants. We use # to represent a signed constant and ## to represent an unsigned constant, *i.e.*, `char16 x = -9#16;` defines -(9) to be signed 16-bit number. We issue a warning if one of these operators is not used with a constant, as it may produce incorrect results with negative numbers.

Extending and Reducing Operators: In order to reduce the size of some circuits, we provide a multiplication operator ($**$) that takes in two X-bit numbers and produces a $X*2$-bit number, and operators for division (//) and modulus (%%) that take in a dividend of size X, quotient of size $ceiling((X+1)/2)$, and returns a $ceiling((X+1)/2)$-bit result. We note the modulus and division operator only work correctly when the result can completely fit in the quotient. These circuits are much smaller than in the case of using X to X-bit operators when, in the case of multiplication, a $X*2$-bit result is desired.

More Than Two Parties: Our compiler allows more than two parties in the computation unlike the other compilers we examined. Adding additional parties to the computations can be useful to declare different types of output.

```
if(x) { if(y) { } else { } }
else { if(z) { } else { } }
```

Figure 11: Twice nested *if* statements. There are 8 possible combinations as x, y, and z can either be 0 or 1.

Appendix C. Frigate Validation Details

To practically validate a compiler, we must check all possible ways that each statement can output a sub-circuit and the ways in which data can flow from the beginning to the end of the program (*e.g.*, when the data is encapsulated in variables, when it is used in control structures, etc.). While daunting, the task is made simpler by realizing that each operator and control structure can only be output in a finite number of ways, *i.e.*, an *if/else* statement has 2 possibilities: it is either the first *if/else* statement or is nested under at least one other *if/else* statement.

We perform the tests outlined in Section 3. At the conclusion of these tests we have covered most, if not all, the state space in Frigate. We have tested (1) the correctness of each mini-circuit an operator uses, (2) all the ways in which data can populate each operator, (3) the base and nested rule for each construct (*if* statements, *for* loops, and arrays declarations), (4) common edge cases, and (5) unique constructs to Frigate's input format. Some tests, like verifying whether a file is correctly included, were performed by printing out the AST and not by compiling and executing the program.

For each type of test, we test a variety of positive (correct) and negative (incorrect) results with emphasis on edge cases. For instance, for the addition operator we test the following (1) Does the operator correctly perform with all possible unsigned 8-bit input combinations? (2) Does the operator correctly perform with all possible signed 8-bit input combinations? (3) Does adding two different types of the same length give a warning? (4) Does adding two different types of different lengths give an error?

Operators: The first tests on operators examine whether the sub-circuits, or templates, for each operator (addition, subtraction, etc.) are correct. For each of these operations, we constructed a large program to test all possible input combinations of 8-bits. This involves 65,536 tests for each binary operator for both signed and unsigned values.

Once we know the template circuits are correct, we must then show all possible types of data that can be entered into the template work as well. These types are: (1) constants, (2) variables, or (3) results from an expression. Once these tests pass, we know the operator can correctly input the different possible types of data.

Control Structures: Once each operator is shown to be correct, we know any other errors found will not be from the primitive operators but from the control structures. There are four control structures in Frigate we must test: functions, *if/else* statements, *for* loops, and procedures.

For each control structure, we check every way in which it can be output. Conditional *if/else* statements can

	Time(s)	All Gates	Non-XOR	Time(s)	All Gates	Non-XOR
Program	Frigate			PCF		
Hamming 1000	0.0067 ± 7%	17,829	4,691	5.1 ± 1%	21,970	4,882
Hamming 16384	0.053 ± 4%	294,737	77,273	6.95 ± 0.8%	391,683	96,117
Mult 256	0.038 ± 5%	391,171	131,330	54.2 ± 0.7%	1,659,808	400,210
Mult 4096	7.94 ± 0.5%	100,630,531	33,558,530	63.7 ± 0.9%	364,605,460	89,444,609
Matrix Mult 5	0.011 ± 2%	372,377	128,252	60.2 ± 0.4%	433,475	127,225
Matrix Mult 16	0.17 ± 1%	12,201,986	4,202,498	68.8 ± 0.4%	14,308,864	4,186,368
AES	0.009 ± 10%	34,889	10,383	0.48 ± 5%	38,260	12,578
RSA 256	0.306 ± 0.5%	942,210,819	202,441,218	272 ± 0.6%	673,105,990	235,925,023
RSA 512	1.08 ± 0.2%	7,526,940,163	1,615,070,210	275 ± 0.8%	5,397,821,470	1,916,813,808
Program	CBMC			KSS		
Hamming 1000	0.71 ± 2%	54,233	18,906	2.2 ± 1%	20,493	4,641
Hamming 16384	1.16 ± 0.8%	910,495	290,728	4.21 ± 0.8%	370,110	88,952
Mult 32	0.48 ± 1%	6,223	1,741	0.34 ± 6%	15,935	5,983
Mult 256	9,800* ± 5%	5,880,833	2,264,860	17 ± 2%	1,044,991	391,935
Matrix Mult 5	1.8 ± 2%	795,988	223,720	32 ± 2%	1,968,452	746,177
Matrix Mult 16	1,500* ± 7%	26,182,494	7,251,991	1900 ± 5%	64,570,969	24,502,530
AES	0.60 ± 4%	35,607	11,469	0.71 ± 1%	49,912	15,300
RSA 256	-**	-	-	14,000 ± 4%*	928,671,864	315,557,288
Program	Obliv-C (time and # of non-XOR gates)			ObliVM (time and # of non-XOR gates)		
Hamming 1000	0.916 ±4%	7,719		0.148 ±7%	1,989	
Hamming 16384	0.962 ±6%	126,945		0.150 ±5%	32,752	
Mult 256	0.978 ±3%	95,296		0.837 ±3%	523,776	
Mult 4096	0.927 ±4%	146,292,736		0.844 ±4%	134,209,536	
Matrix Mult 5	0.942 ±5%	127,969		0.851 ±5%	653,125	
Matrix Mult 16	0.919 ±5%	4,194,273		0.862 ±5%	139,591,680	
AES	1.549 ±4%	385,056		0.198 ±8%	61,227	
RSA 256	1.031 ±5%	169,993,375		-	-	
RSA 512	1.083 ±6%	3,525,298,575		-	-	

TABLE 4: This table shows the compile time in seconds, the amount of total gates, and the amount of non-XOR gates. Note that for CBMC and KSS, we ran Mult 32 and Mult 256 instead of Mult 256 and Mult 4096.
All tests were ran 10 times unless otherwise noted: * tests ran 3 times, ** we stopped compilation after 6 hours.

be checked for correctness by performing an exhaustive search up to depth 2 (*i.e.*, test all 8 possible cases of the conditional output as shown in Figure 11). The unique ways to output the *if/else* statement occur within the first conditional, while the depth-2 *if/else* statement must be combined with its parent conditional. If the nested *if/else* conditionals combine correctly at depth 2 then by induction, it will work for subsequently nested conditionals as well. Each *if/else* statement should be tested for when the guard values are dependent on user input as well as when they are not dependent on user input.

It is relatively simple to validate the correctness of *for* loops when they are only nested under another *for* loop by checking whether they output the circuit the correct number of times. When they are used under *if/else* statements, problems can arise depending on how the loop variable is scoped and whether the loop variable's result will be labeled *UNKNOWN*, meaning the result is based on user input due to the *if* statement, or whether it will be labeled as a 0 or 1 value. We test to depth 2 in case there is an external state used by the compiler that may prevent nested *for* loops from working correctly under *if/else* statements.

Functions also have a finite number of possible states to test. Our procedure for carrying out this testing was as follows. (1) Test the function call operator, where a function call is treated as any other operator that takes in any number of operands (parameters) and returns a single operand. We test different possible combinations of parameters up to length 2, as that is where data no longer acts in a unique way. (2) Function definitions need to be tested to ensure different types of return variables (array, struct, int, uint) work correctly. (3) Test two of the same function call in an operand with different results (*e.g.*, `addX(3,4) + addX(5,6)`). It is possible that the results of the first call may be overwritten by the second call. (4) Test that parameters can be used inside of the functions.

The correctness of procedures reduces to a simple question: *is each variable composed of the exact same wires every iteration?* We know that variables will use the same free wires if the wire pool is sorted before each iteration.

Complete Validation Set: We performed a much more extensive suite of tests on Frigate than we performed on the other compilers. We went looking for where we believed we would find errors in Frigate and in some cases we found errors during validation. Other than the millions of arithmetic tests we performed, our validation test set has more than 17,000 tests. This set contains the above test cases and has many arithmetic tests. Most of the tests were generated using various Java programs, *i.e.*, we could template the problem of nested if statements and then generate all possible combinations.

Once we created our set of validation set program, we integrated it to be performed every time `make` is called. This way, we can see if any changes we made to the compiler broke a test case (and this has been useful).