On Developing Privacy-Preserving Compilers

Yu Yu, and Jussipekka Leiwo, and Benjamin Premkumar

Nanyang Technological University, School of Computer Engineering, Nanyang Avenue, Singapore

Summary
In this paper, we discuss whether or not it is possible to execute a program on an untrustworthy computer without revealing anything substantial. We simulate this task by developing a compiler that transforms a program *p* to an equivalent circuit format *GC*, which can be executed remotely on an untrustworthy computer by taking as argument encrypted input and producing encrypted output. The whole computation is totally hidden from the computer. The design of the compiler is detailed. With our compiler, polynomial-time programs can be efficiently converted to polynomial-size Boolean circuits.

Key words:
Compiler design, private computation, Boolean circuit, information hiding.

Introduction

1.1 Problem Formalization

Alice has a private program p and she wants to compute p with some private input x but lacks resources to do it. Bob is a powerful computer and is willing to help Alice. Alice hopes that p can be executed by Bob in such an oblivious way that nothing substantial about p, x and p(x) is disclosed to Bob.

1.2 Related Work

Abadi, Feigenbaum, and Kilian [2] described computing with encrypted data (CED) as follows: Alice wishes to know $f(x)$ for some x but lacks power to compute it. Bob has the power to compute f and is willing to send $f(y)$ to Alice if she sends him y, for any y. Alice transforms x into an encrypted instance y, obtains $f(y)$ from Bob and infers $f(x)$ from $f(y)$ in such a way that B cannot infer x from y.

If such an encryption scheme exists, f is considered encryptable. They found that problems such as Discrete Logarithm and Primitive Root are encryptable. However, they did not propose any encryption scheme for general function f. Abadi and Feigenbaum [1] proposed a circuit evaluation protocol for CED. However, their method cannot evaluate AND gates non-interactively. Sander et al. [9] presented an AND-PH to solve this problem, but their method only allows evaluation of log- depth circuits.

Sander and Tschudin [8] proposed a solution to compute with encrypted functions (CEF): Alice has a private function f. Bob has an input x. Alice wants Bob to compute f (x) without revealing anything substantial about f. Their scheme only allows encryption of polynomials. Loureiro [6] presented another scheme which allows encryption of a general function f with small inputs. This approach, however, fails to meet our goal since a non-trivial program usually has an input of at least hundreds of bits.

Above approaches attempt to find universal encryption schemes, either for function f or for input x, that can be used repeatedly with provable privacy. Nevertheless, none of them seems to provide a satisfactory solution for our scenario due to the lack of generality.

In software industry, a lot of commercial software (i.e. shareware) will be packed (compressed or encrypted) to prevent reverse engineering and cracking. Figure 1 shows how an executable is packed. The main body of the code segment is encrypted and thus cannot be analyzed by static dis-assemblers. However, when it is executed, the whole image of the executable file will be loaded into memory and the encrypted code will be decrypted by the decryption routine (located at the end of the image) prior to the execution. Therefore, we can use a debugger to dump these codes (in plain text) to a new executable right after the decryption is done. These tricks are also used by some viruses to hide themselves from detection of anti-virus software. However, since these tricks have no cryptographic foundations, they are used to prevent reverse engineering only for a limited period of time. Another related technique is program obfuscation, namely, a program is rendered unintelligent to reverse engineers but still remain its original functionality. Unfortunately, it has been proved that universal obfuscators do not exist [4].

Manuscript reviced March 19, 2006.

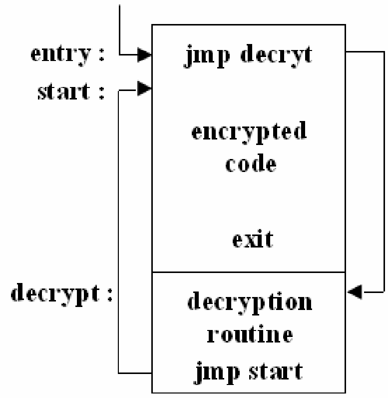


Fig. 1 A packed executable file.

1.3 Our Solution

We develop a compiler that on input a user-written C-style source code *p*, produces as output the encoding of a garbled circuit *GC*. We also develop a virtual machine on which *GC* can be run obliviously.

2. Solution Overview

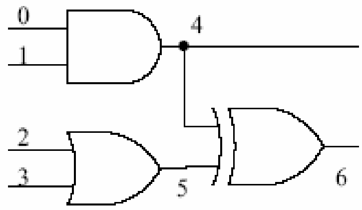
The compiler can be viewed as two subroutines, a program-circuit transformer and a circuit-encryptor, where the former transforms *p* into a Boolean circuit *C* and the latter encrypts *C* to produce *GC*, which can be executed obliviously by an untrustworthy party.

2.1 Boolean Circuits

Informally, a Boolean circuit is a directed acyclic graph with internal nodes characterized by Boolean gates (e.g., gates numbered 4 through 6 in Fig. 2). Nodes with no incoming edges are called circuit-inputs (e.g., gates numbered 0 through 3 in Fig. 2) and those with no outgoing edges are circuit-outputs. The size of a Boolean circuit is the number of its gates. The functionality of a gate can be expressed with truth tables, e.g., for gate of fan-in 2, its functionality $g(a, b)$ whose inputs are a and b can be represented using truth table $[g(0,0), g(0,1), g(1,0), g(1,1)]$.

2.2 Possibility of Transforming Polynomial-Time Programs to Polynomial-Size Circuits

Since the von Neumann architecture is the most prevailing computer architecture, we assume that programs correspond to micro-instructions that can be executed on a von Neumann computer. Such a computing device has a counter,



Input	4	(0,1,2,3)		
Output	2	(4,6)		
0	INPUT		[0, 1]	
1	INPUT		[0, 1]	
2	INPUT		[0, 1]	
3	INPUT		[0, 1]	
4	GATE	(0, 1)	[0, 0, 0, 1]	OUTPUT
5	GATE	(2, 3)	[0, 1, 1, 1]	
6	GATE	(4, 5)	[0, 1, 1, 0]	OUTPUT

Fig. 2 An example of Boolean circuit.

a memory, and a CPU that can perform the following micro-instructions [3]: Load (from a memory location to a register), Store (from a register to a memory location), Add, Complement, Jump, JumpZ (for conditional branching) and Terminating. Informally, a family of programs $\{p_n:\{0,1\}^n \rightarrow \{0,1\}^m\}_{n\in N}$ is polynomial- time computable if there exists a polynomial *poly* such that the number of micro-instructions processed by the CPU before p_n terminates is at most *poly*(*n*).

We can establish the possibility of converting polynomial-time programs to polynomial-size Boolean circuits with the following steps. First, it is well-known (see e.g. [3, Theorem 1.3]) that each of the above micro-instructions can be simulated by a Turing machine in polynomial time and consequently problems solvable by a von Neumann computer in polynomial time can also be solved by a Turing machine in polynomial time. Second, Goldreich [5] constructed a Boolean circuit that simulates the run of a Turing machine *M* on input $x\in\{0, 1\}^n$ with a circuit size quadratic in *TM* (*n*) (the running time of *M* on input of length *n*), namely, problems solvable by Turing machine in polynomial-time can be solved using polynomial-size Boolean circuits.

2.3 Program-Circuit Transformer

Although it is theoretically possible to convert polynomial-time programs to polynomial-size circuits, the approach in Sect. 2.2 is inefficient in that the conversion cannot be done directly. Malkhi et al. [7] implemented a compiler that can represent simple programs (e.g., the Millionaire problem and the Private Information Retrieval problem) by Boolean circuits, but their compiler only supports two arithmetic operations, addition and subtraction, but complicated programs require multiplication and division. To solve this problem, we develop a compiler independently and ours is more powerful in that it supports multiplication, truncating division, rounding division and modular arithmetic. The Boolean gates generated by our compiler have fan-in bounded by 3. The BNF grammar defined by our compiler is similar to that of the *C* language, for example, we can simulate the micro-instruction "jumpZ" by "IF" statements. The design of such a compiler is not a trivial task because the object code is a Boolean circuit that is totally different from micro-instruction in that it does not have branching when executed.

Our compiler supports three data types: Boolean, signed integer and unsigned integer. In contrast to computers whose CPUs can only process data of fixed length, we can declare an integer to be of an arbitrary constant length, namely, the cost of solving a family of problems is measured by the size of input in a uniform manner. For example, let $\{p_n:\{0,1\}^{2n}\rightarrow\{0,1\}^n\}_{n\in N}$ be a family of programs that take two n-bit-long integers as argument and produce their sum, it is obvious that the solution on computers is non-uniform because the data must be partitioned to fixed-length (e.g. 32-bit) to be processed by CPU in case of large n, nevertheless, with our compiler, we only need to define two input integers A_n and B_n , and then write in the source code of p_n as

return($A_n + B_n$);

And the compiler will generate a circuit of $2n$ Boolean gates that computes the same function as p_n does.

We discuss informally how many Boolean gates general polynomial-time program p_n needs. When p_n is executed, it will terminate after at most *poly*(n) basic operations, which includes logical operations, arithmetic operations, comparisons, value assignments, etc. As depicted in Table 1, each basic operation corresponds to no more than $3mn^0 +6m+n^0$ Boolean gates, where m and n^0 are bounded by a fixed polynomial of n. Thus, the number of Boolean gates generated for each basic operation is also bounded by *poly'*(n) and the resulting number of Boolean gates *poly*(n)×*poly'* (n) is still polynomial in n.

2.4 Circuit Encryptor

After converting a program to the functionally equivalent Boolean circuit, we can encrypt the circuit using Yao's method [10] such that the encrypted circuit can be executed by an untrustworthy party (e.g., a remote PC) without revealing anything substantial to it.

Table 1: The number of Boolean gates needed for each basic operation, where cost1 and cost2 are costs for unsigned operands and signed ones respectively.

Operation	Operands	Cost 1 (unsigned)	Cost 2 (signed)
Logical operations	A_{n^0}, B_{n^0}	n^0	n^0
Addition/Subtraction	A_{n^0}, B_{n^0}	$2n^0$	$2n^0$
Multiplication	A_m, B_{n^0}	$3mn^0$	$3mn^0+m+n^0-4$
Truncating division	A_m, B_{n^0}	$3mn^0+3m$	$3mn^0+4m-n^0-5$
Modular arithmetic	A_m, B_{n^0}	$3mn^0+3m$	$3mn^0+2m+n^0-5$
Rounding division	A_m, B_{n^0}	$3mn^0+5m+2n^0+2$	$3mn^0+6m+n^0-7$
Comparison	A_{n^0}, B_{n^0}	n^0	n^0
Value assignment	A_{n^0}, B_{n^0}	n^0	n^0

3. Compiler Design

3.1 Data Types and Data Declaration

The compiler supports three data types: Boolean, signed integer and unsigned integer. A Boolean is a 1-bit-long variable that is mostly used in selection statements. Signed integers and unsigned integers are variables that can be declared to be of arbitrary constant (no less than 2) length. Unsigned integers are internally represented as base 2. Thus, the value of unsigned integer A_n, with the representation $a_n\cdots a_1$, is simply its base 2 value, namely,

$$\sum_{i=1}^{n} a_i \times 2^{i-1}$$

Signed integers are represented using two's complement, e.g., the value of $B_n = b_n\cdots b_1$ is

$$-2^{n-1}\times b_n+\sum_{i=1}^{n-1} b_i \times 2^{i-1}$$

We can also declare constants without necessarily specifying their data types. For example,

unsigned int (30) *A*;
bool *b*;
sigined int (50) *C* ;
const *D*=15;

are a list of data declarations where *A*, *b*, *C* and *D* are declared to be a 30-bit-long unsigned integer, a Boolean, a 50-bit-long signed integer and constant 15 respectively.

3.2 Language Syntax

The language acceptable by the compiler is defined using Backus-Naur Form (BNF) that consists of a set of a production rules. A production rule states that the symbol (i.e. non-terminal) on the left-hand side of the ":" must be replaced by one of the alternatives on the right hand side, where the alternatives are separated by "|". For example,

```
        symbol :
                 alternative1
                 | alternative2
                 ...
```

With production rules, programmers can write programs that can be recognized by the compiler. The recognition is done by applying the production rules in reverse (i.e. LL(1) grammar). That is, the compiler parses the input program terminal (basic unit of strings that make sense to the compiler, e.g. IF, FOR and ';') by terminal, chooses the right rule by looking at only the current terminal on the input and takes the corresponding action. The grammar defined by the compiler can be summarized with the following production rules:

```
statements : statement
            |statements statement

statement : variable ':='expression ';'
          | RETURN expression ';'
          | IF '(' bool ')'
            '{' statements '}'
          | IF '(' bool ')'
            '{' statements '}'
           ELSE
            '{' statements '}'
          | FOR variable :'='
            expression TO expression
            '{' statements '}'

expression : variable
          | constant
          | '(' expression ')'
          | NOT expression
          | expression
            logical_operator expression
          | expression
           arithmetic_operator
expression

bool : TRUE
     |   FALSE
     |   expression compare expression
     |'(' bool ')'
     | bool logical operator bool
```

where the rules are oversimplified for the sake of demonstration. For example, operators (e.g. +, −, ×, ÷) are considered to be of the same operator precedence and there is a reduce-shift conflict when parsing "IF" and "IF-ELSE" statements, but all these problems can be solved by introducing detailed rules.

3.3 Operations between Expressions

With the production rules, we know that an (logical or arithmetic) operation between two expressions will be reduced to a new expression. The compiler will generate Boolean gates for this new expression such that it can be further referred to by other operations. We first show how the operations between unsigned integer expressions are implemented by the compiler and then reduce the operations between signed integer expressions to the unsigned analogue. We assume that A_m (resp., B_n) is an m-bit-long (resp., n-bit-long) integer expression with binary representation $a_m \cdots a_1$ (resp., $b_n \cdots b_1$). Of course, each label a_i (resp., b_j) corresponds to a circuit-input, or a Boolean constant, or an output of some Boolean gate generated by the compiler.

Logical operators can be either unary (e.g. NOT) or binary (e.g. AND, OR, XOR, etc) and the operands can be Booleans and integers. For uniformity, we treat Boolean as 1-bit-long integer and let "*" be the logical operator, then the gates generation algorithm can be described using the following pseudo-code:

```
program Logic_Op (A_n, B_n/-, *)
    for i = 1 to n do
        if * = NOT
            c_i ← gate(a_i) = ā_i
        else
            c_i ← gate(a_i, b_i) = a_i*b_i
        end if
    end for
    result = c_n···c_1
end program
```

where $c \leftarrow$ gate(a, b) means generating a Boolean gate whose inputs are a and b and whose output are labeled by c. Labels can be reused, e.g., $a \leftarrow$ gate(a, b) indicates that gate with inputs a and b is generated and label a is reallocated to the output of the resulting gate. Addition/subtraction between unsigned integers is handled as follows:

```
program Unsigned_Add/Sub (A_n, B_n)
    c_1 ← 0/1
    for i = 1 to n do
        s_i ← gate(a_i, b_i, c_i)
            = a_i⊕b_i⊕c_i / a_i⊕b̄_i⊕c_i
        c_{i+1} ← gate(a_i, b_i, c_i)
            = carry(a_i⊕b_i⊕c_i)/carry(a_i⊕b̄_i⊕c_i)
    end for
    sum = c_{n+1}s_n···s_1 / difference = c̄_{n+1}s_n···s_1
end program
```

where carry($a \oplus b \oplus c$) = $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$ and $2n$ Boolean gates are generated. Multiplication can be implemented by invoking the above subroutine, namely,

```
program Unsigned_Mul (A_m, B_n)
    s_{m+n}···s_1 ← 0
    for i = 1 to n do
        for j = 1 to m do
            c_j ← gate(a_j, b_i) = a_j b_i
        end for
        s_{i+m}···s_i ← Add(s_{i+m-1}···s_i, c_m···c_1)
    end for
    product = s_{m+n}···s_1
end program
```

Thus, multiplication needs at most 3*mn* Boolean gates. The Boolean gates of rounding division ”DivR”, truncating division ”DivT” and modular arithmetic ”Mod” can be generated using the following subroutine:

```
program Unsigned_DivR/DivT/Mod(A_m, B_n)
    r_{m+n}···r_1 ← \overbrace{0···0}^{n} a_m···a_1
    for i = m to 1 do
        \bar{q}_i t_{n+1}···t_1 ← Sub(r_{i+n}···r_i, 0b_n···b_1)
        for j = 1 to n+1 do
            r_{i+j-1} ← gate(q_i, t_j, r_{i+j-1})
                      = (q_i ∧ t_j) ∨ (\bar{q}_i ∧ r_{i+j-1})
        end for
    end for
    if DivR
        \bar{q}_0 t_{n+1}···t_1 ← Sub(r_n···r_1 0, 0b_n···b_1)
        q_{m+1}···q_1 ← Add(q_m···q_1, \overbrace{0···0}^{m-1} q_0)
    end if
    quotient = q_m···q_1 / remainder = r_n···r_1
end program
```

Now we consider the arithmetic operations in case of signed integer operands. The addition and subtraction of signed integer expressions is similar to their unsigned counterparts since we use two’s-complement integer representation for signed integers. Other operations can be implemented by invoking their signed analogue as follows:

```
fuction 2's_complement (A_m, b)
    /*If b=1, 2's-complement A_m;
     otherwise do nothing.*/
    c_1 ← b
    for i = 1 to m
        c_{i+1} ← gate(a_i, b, c_i)
                = b ∧ carry(\bar{a}_i ⊕ c_i)
        a_i ← gate(a_i, b, c_i)
            = (b ∧ (\bar{a}_i ⊕ c_i)) ∨ (\bar{b} ∧ a_i)
    end for
    2's-complement = a_m···a_1
end function
```

```
program Signed_Mul/DivR/DivT/Mod(A_m, B_n)
    s_a ← a_m
    s_b ← b_n
    s ← gate(s_a, s_b) = s_a ⊕ s_b
    A_{m-1} ← 2's_complement(a_{m-1}···a_1, s_a)
    B_{n-1} ← 2's_complement(b_{n-1}···a_1, s_b)
    r_o···r_1 ← Unsigned_Mul/DivR/DivT/Mod(
            A_{m-1}, B_{n-1})
    r_o···r_1 ← 2's_complement(r_o···r_1, s)
    result = s r_o···r_1
end program
```

3.4 Comparisons between Expressions

The compiler will generate a Boolean indicating the result of comparison between expressions. There are six comparison operators as depicted in Table 3, where (*An* ==*Bn*, *An*!=*Bn*), (*An*>*Bn*, *An*<=*Bn*) and (*An*<*Bn*, *An*>=*Bn*) are complementary pairs and An >Bn can be viewed as Bn<An . Thus, it suffices to show the pseudo-code of An ==Bn and An<Bn, which is as follows:

```
program Same(A_n, B_n)
    c_1=1
    for i = 1 to n
        c_{i+1} ← gate(a_i, b_i, c_i)
                = c_i ∧ (a_i ⊕ b_i ⊕ 1)
    end for
    result = c_{n+1}
end program
```

```
program Less_Than(A_n, B_n)
    c_1=1
    for i = 1 to n-1
        c_{i+1} ← gate(a_i, b_i, c_i)
                = carry(a_i ⊕ \bar{b}_i ⊕ c_i)
    end for
    if A_n and B_n are unsigned expressions
        \bar{c}_{n+1} ← gate(a_n, b_n, c_n)
                = carry(a_n ⊕ \bar{b}_n ⊕ c_n)
    else
        c_{n+1} ← gate(a_n, b_n, c_n)
                = (a_n ∧ \bar{c}_n) ∨ (\bar{b}_n ∧ \bar{c}_n) ∨ (a_n ∧ \bar{b}_n ∧ c_n)
    end if
    result = c_{n+1}
end program
```

3.5 Selection Statements and Value Assignments

The two forms of selection statements supported by our compiler are ”IF(<bool>)-<statements>”
and
”IF(<bool>)-<statements>-ELSE-<statements>”,
where ”bool” is a label of the Boolean expression in the parentheses. When the selection statements are executed

on computers, the control is passed to the statement following ”IF” if the ”bool” is nonzero, otherwise it is passed to the second statements (if any). If we generate Boolean gates in this way, the resulting circuits will have conditional branches and the flow of control might vary for different inputs. This is non-oblivious since the flow of control will reveal sensitive information (e.g. value of ”bool”) even if the circuit is evaluated in its encrypted format. Therefore, our compiler generates Boolean gates that have uniform control flow when evaluated, namely, the resulting circuit is evaluated sequentially without a single gate to be skipped. In fact, this is not hard to achieve since most operations in selection statements can be done identically as they are in statements outside ”IF” with one exception being the operation of assigning value. This is because that the compiler generates new Boolean gates to store intermediate results and the values of variables are not changed unless value assignments happen. Based on the fact that the value of a variable will be updated only if the ”bool” is non-zero, we initializes a stack with only one item ”TRUE” (i.e. 1) on its top at the start of compilation. The stack operations are as follows:

```
data[ ]
//"[ ]"indicates a vector with
  t the number of items in the
  stack program Init_Stack()
     data[0] ← TRUE
     t ← 1
end program
program push (bool)
     data[top] ← bool
     t ← t+1
end program
program pop ()
     t ← t-1
end program
program top()
     return data[t-1];
end program
```

where ”bool” is the label passed to sub-routine push(-). We also describe the actions taken by the compiler when it enters/leaves the statements of IF and ELSE whose Boolean expression in the parentheses is labeled by ”*b*”, namely,

```
program enter_if_stmt/enter_else_stmt (b)
    a ← top()
    c ← gate(a,b) = b∧a / b̄∧a
    push (c)
end program

program leave_if_stmt/leave_else_stmt (b)
    pop()
end program
```

Thus, the pseudo-code of ”An:=Bn”, whether in selection statements or not, can be uniformly written as:

```
program assign_value (A_n, B_n)
    c ← top()
    for i = 1 to n
        a_i ← gate(a_i, b_i, c)
            = (c∧b_i) ∨ (c̄∧a_i)
    end for
    A_n = a_n ··· a_1
end program
```

3.6 Iteration Statements

The iteration statement supported by the compiler is
FOR <variable> :=
<expression1> TO <expression2>
{ <statements> }

where <statements> are repeatedly executed unless ”<variable>” exceeds the range specified by the two expressions. However, a Boolean circuit is a directed acyclic graph and thus Boolean gates cannot be reusable. To solve this problem, the compiler treats the iteration statement as a macro and unrolls it during the preprocessing stage to produce:

$$
\begin{array}{c}
\langle\text{variable}\rangle := \langle\text{expression1}\rangle \\
\langle\text{statements}\rangle \\
\langle\text{variable}\rangle := \langle\text{variable}\rangle + increment \\
\langle\text{statements}\rangle \\
\vdots \\
\langle\text{variable}\rangle := \langle\text{expression2}\rangle \\
\langle\text{statements}\rangle
\end{array}
$$

where <expression1> and <expression2> should be constant expressions and increment is 1 if <expression1> is less than <expression2> and is −1 otherwise. The unrolled program is functionally equivalent to the corresponding iteration statement and it can be easily converted to Boolean gates. Since our compiler requires that the iteration number must be determined at compile time, it does not support statements such as
WHILE (<expression>) { <statements> }

where <statement> is executed repeatedly as long as the value of the <expression> remains true. Nevertheless, we can solve this problem by rephrasing it as

```
FOR i := 1 TO max{
    IF(<expression>){
        <statements>
    }
}
```

where max is the upper bound of number of iterations that the ”while-statement” cannot exceed. For a polynomial-time program, max is still bounded by a polynomial, e.g., a bubble sort program has at most $n(n-1)/2$ iterations, where n is the number of inputs to be sorted.

3.7 Return Statements

Usually a program or a function will halt after a ”RETURN” statement and will return a value (if any) to the the environment that called it, but in our case, ”RETURN <expression>;” only indicates that <expression> is a circuit-output. Thus, the compiler will mark the labels of the gates that correspond to <expression> as final outputs and continue parsing the program.

4. Concluding Remarks

We have developed a privacy-preserving compiler that maps a polynomial-time program to a polynomial-size circuit. Such a compiler is useful in cryptography and private computation as it allows an untrustworthy to execute a program without revealing anything substantial to him.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] M. Abadi and J. Feigenbaum. Secure circuit evaluation: A protocol based on hiding information from an oracle. *J. Cryptol.2*, pages 1–12, 1990.

[2] M. Abadi, J. Feigenbaum, and J. Kilian. On hiding information from an oracle. *J. Comput. Syst. Sci.*, 39:21–30, 1989.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vad- han, and K. Yang. On the (im)possibility of obfuscating programs. In *Proc. CRYPTO ’2001*, pages 1–18. Springer-Verlag, 2001.

[5] O. Goldreich. Introduction to complexity theory – lecture 2: NP- completeness and self reducibility. The Weizmann Institute of Science, Israel, 1999. (http://www.wisdom.weizmann.ac.il/˜oded/cc99.html).

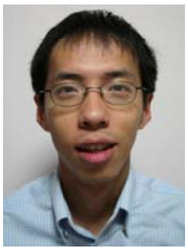
[6] S. Loureiro. *Mobile Code Protection*. PhD thesis, Ecole Nationale Superieure des Telecommunication, 2001.

[7] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Proc. Usenix Security 2004*, 2004.

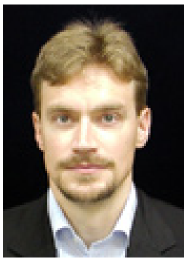
[8] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security, LNCS*, volume 1419, pages 44–60, 1998.

[9] T. Sander, A. Young, and M. Yung. Non-interactive cryptocomputing for NC1. In *In Proc. 40th Annual IEEE Symp. Found. Comput. Sci.*, page 554, 1999.

[10] A. C. Yao. How to generate and exchange secrets. In *Proc. 27th Annual FOCS*, pages 162–167, 1986.



Yu Yu received his B.S. degree in Computer Science from Fudan University, Shanghai, in 2003. He is now a PhD candidate in Computer Science at Nanyang Technological University, Singapore.



Jussipekka Leiwo received his MSc and PhD degrees from University of Oulu and Monash University respectively. He is now an assistant professor at School of Computer Engineering, Nanyang Technological University, Singapore.

Benjamin Premkumar received his MSc and PhD degrees from North Dakota State University and university of Illinois respectively. He is now an associate professor at School of Computer Engineering, Nanyang Technological University, Singapore.