Language Based Information Routing Security: Policy Enforcement

George Oreku [1], Jianzhong Li[1], and Fredrick Mtenzi[2]
[1]Department of Computer Science and Engineering, Harbin Institute of Technology, China
[2] School of Computing, Dublin Institute of Technology, Ireland

Abstract: *Languages-based security promises to be a powerful tool with which provably secure routing applications may be developed. Programs written in these languages enforce a strong policy of non-interference, which ensures that high-security data will not be observable on low-security channels. The information routing security proposed aim to fill the gap between representation and enforcement by implementing and integrating the divers security services needed by policy. Policy is enforced by the run-time compiler and executions based mechanism to information violating routing policy and regulation of security services. Checking the routing requirements of explicit route achieves this result for statements involving explicit route. Unfortunately, such classification is often expressed as an operation within a given program, rather than as part of a policy, making reasoning about the security implications of a policy more difficult. We formalize our approach for a C++ like language and prove a modified form of our non-interference method. We have implemented our approach as an extension to C and provide some of our experience using it to build a secure information routing.*

Keywords: *Security policy, routing, condition statements*

Received January 5, 2007; accepted November 6, 2007

1. Introduction

An information routing policy is a security policy that describes the authorized paths along which that information can route. Each model associates a label, representing a security class, with information and with entities containing that information. Each model has rules about the conditions under which information can move throughout the system. Historically communication security policies have been always crafted for the specific systems they support [18, 7] we find that language provided the rudimentary tools to achieve low-level security goals and its extension were necessary to formulate and enforce application policy.

These languages provide a means of provably enforcing a security policy in a broader sense. A current technique for enforcing security routing relies on so called best practices [11] which include simplistic techniques (such as password, TCP, authentication, rout filter, and private addressing) to mitigate the most rudimentary vulnerabilities and threats.

Security-typed languages annotate source code with security levels on types [17], such that the compiler can statically guarantee that the program will enforce noninterference [5]. In a broader sense, these languages provide a means of provably enforcing a security policy. Theoretical models for security-typed languages have been actively studied and are continuing to evolve. For example, researchers are extending these models to include new features, such as exceptions, polymorphism, objects, inheritance, side-effects, threads, encryption, and many more [13].

To address this lack of practical experience, we build a realistic application in a security-typed language. We sought to discover whether this C++ secure language programming could hold up to its promise of delivering real-world applications with strong security guarantees. Two key criteria we used for defining "real-world" were that (1) the application should interact with other non-security-typed, networked components while still maintaining the security policy of its data, and (2) the security policy should be easily re-configurable such that the application could be of general use (not just in a military, MLS setting, but also in a corporate setting, for example). We conducted this through a security-typed variant of C++ codes examples and definitions on routing processes. Throughout, we reflect on the examples and definitions of language-based security codes

A principal result of this study is that we succeeded in developing a real-world application for which we can easily assess that there is no information leakage beyond what is allowed by a clear, user-defined, high-level policy. We found that while language tools were robust and expressive, additional development and runtime tools were necessary. We provide a critical evaluation of the C language through examples and definitions, highlighting its effectiveness at carrying out the promised security goals, the difficulties involved in using it and the ways in which it still needs improvement.

However to approach the problem, we use C's type class mechanism to give an interface for security lattices. Programs written in the embedded language can be parameterized with respect to this interface. Moreover, the embedded language can easily be given security-specific features such as a declassification operation or run-time representation of privileges for access-control checks. In both cases, we make use of C's strong type system to guarantee that the abstractions enforcing the security policies are not violated. This encapsulation means that it is not possible to use the full power of the C language to circumvent the information-routing checks performed by the embedded language, for example what are reasonable information routing policies? For each variable x, define $\underline{x}$ to be its information routing class. An information routing policy restricts flow between certain classes and is a relation on the set of information routing classes. (Think of classes as: top secret, secret, confidential, *etc.*) A policy might be: no information routing from secret to unclassified. Why is the lattice assumption useful? Note that the lub and glb properties come "for free." It is always possible to add elements to a lattice (top and bottom, for example) to satisfy the lub and glb requirements. It turns out that having a lattice will allow us to compute some things very efficiently. Recall the example where *x* and *y* are natural numbers and we assign $z := x + y$. We would like to analyze the expression $x + y$ (as opposed to examining each individual variable) in testing to see if execution of $z: = x + y$ should be allowed. The existence of a lattice implies: if $\underline{x_1} <= \underline{y}$, $\underline{x_2} <= \underline{y}$, . . ., $\underline{x_n} <= \underline{y}$ then there exists some $\underline{x}$ where $\underline{x} = \underline{x_1}$ lub $\underline{x_2}$ lub $\underline{x_3}$ . . . $\underline{x_n}$ and $\underline{x} <= \underline{y}$. Therefore, flows $x_1\text{->}y$, $x_2\text{->}y$, . . .lub $x_n\text{->}y$, all are authorized if and only if $\underline{x_1}$ lub $\underline{x_2}$ lub $\underline{x_3}$ . . . lub $\underline{x_n} <= \underline{y}$. And, to check if a policy is satisfied, it is only necessary to compute one least upper bound, rather than to check a set of <= relations. If the latter computation is expensive, it is useful to only have to do it once.

Our extended view of policy allows us to consider new ways of using context. Security-typed programming language allows the issuers of policy to augment applications through policy specification. We sought to discover whether this tool for secure programming could hold up to its promise of delivering real-world applications with strong security guarantees. In practice, the security policies enforced by program monitors grow more complex both as the monitored software is given new capabilities and as policies are refined in response to attacks and user feedback. This is best illustrated by examples proposed dealing with policy complexity by organizing policies in such a way as to make them compassable. We present a fully implemented compiler and execution-based mechanism that allows security engineers to specify and enforce composeable policies on C++ applications. We also formalize the central workings by defining an unambiguous semantics for our applied language.

1.1. Related Work

Developer tools and programming experience have not evolved in concert with language features. There are currently only two significant language implementations, Flow Caml [14] and Jif [10] and only two applications [1, 10], both written in Jif.

The concept of information-flow control is well established. After the first formulation by Bell and La Padula [2] and the subsequent definition of noninterference [5], Smith, Volpano, and Irvine first recast the question of information flow into a static type judgment for a simple imperative language [5]. The notion of information flow has been extended to languages with many other features, such as programs with multiple threads of execution [16, 18], functional languages and their extensions [6, 12, 19] and distributed systems [8]. For a comprehensive survey of the field, see the survey by Sabelfeld and Myers [21]. Two robust security-typed languages have been implemented that statically enforce noninterference. Flow Caml [14] implements a security-typed version of the Caml language that satisfies noninterference. JFlow [9] and its successor Jif [10] introduce such features as a decentralized label model and run-time principals in an extension to the Java language. Jif is actively in development, with the latest release in June 2006, introducing integrity labels [10].

1.2. Security Challenges, Requirements, and Goals

The security policy we defined at the outset is driven by a range of security goals and requirements, Confidentiality Integrity and Availability (CIA). Based cryptographic traditional security mechanism, such as authentication protocols, digital signature and key management which responsible to keep track of binding keys and assist on establishing mutual trust and secure communications are posing both challenges and opportunities of archiving security goals. Cryptographic in routing protocols gives challenges of difficulties on time synchronizations, dependence complexity of techniques as routing service need to bootstrap themselves (i.e., directories, basic startup operations of management system).Consequence of potential nor loss on investment have been encouraging commercial entities to devote and deploy more secure infrastructure.

No standardized security solutions for most routing technologies, designing secure extension or new protocols are extremely broad. In a long run no single security solutions can address all routing protocols since routing protocols differ in their design even within single routing protocols different security might be required. Platform in which routing protocols are operating is another challenge i.e., More than three orders of magnitude have different exit in the control, different data plane's processing capabilities.

The complexity of and requirements imposed on routing technologies continue to escalate and this will increase the potential vulnerabilities to and consequence of focused routing system attacks [11] Internet Engineering Task Force (IETF) routing protocols security requirements working group gives more discussion on this [15].

2. Information Routing Policy

Information routing policies define the way information moves throughout a system. Typically, these policies are designed to preserve confidentiality of data or integrity of data. In the former, the policy's goal is to prevent information from routing to a user not authorized to receive it. In the latter, information may route only to processes that are no more trustworthy than the data. Any confidentiality and integrity policy embodies an information routing policy. Example*:* the model describes a lattice-based information routing policy. Given two compartments *A* and *B*, information can route from an object in *A* to a subject in *B* if and only if *B* dominates *A*. Let x be a variable in a program. The notation $\underline{x}$ refers to the information routing class of x.

Example: consider a system that uses the model above. The variable x_1 which holds data in the compartment (*TS, {NUC, EUR}), is set to 3. Then* $x = 3$ *and* $\underline{x} = (TS, \{ NUC, EUR \})$.

Intuitively, information routing from an object *x* to an object *y* if the application of a sequence of commands c causes the information initially in x to affect the information in y.

Definition 1: the command sequence c causes a routing of information from x to y if, after execution of c_1 some information about the value of x before c was executed can be deduced from the value of y after c were executed. This definition views information routing in terms of the information that the value of y allows one to deduce about the value in x. For example, the statement $y := x$; reveals the value of x in the initial state, so information about the value of x in the initial state can be deduced from the value of *y* after the statement is executed. The statement $y: = x / z$; reveals some information about x, but not as much as y*:* = *x* statement. The final result of the sequence c must reveal information about the initial value of x for information to route. The sequence

tmp : = x;
y : = tmp ;

has information routing from x to y because the (unknown) value of x at the beginning of the sequence is revealed when the value of y is determined at the end of the sequence. However, no information routing occurs from trap to *x*, because the initial value of trap cannot be determined at the end of the sequence.

Example: consider the statement `x := y + z`; Let y take any of the integer values from 0 to 7, inclusive, with equal probability, and let z take the value *i* with probability 0.5 and the values 2 and 3 with probability 0.25 each. Once the resulting value of x is known, the initial value of y can assume at most three values. Thus, information routes from y to x. Similar results hold for z. For example: consider a program in which x and *y* are integers that may be either 0 or 1. The statement

if x = 1 then y : = 0;
else y : = 1;

does not explicitly assign the value of x to *y.* Assume that x is equally likely to *be 0 or 1.* Then $H(x_s) = 1$. But $H(x_s \ I \ y_t) = 0$, because if y is *0,* x *is 1,* and vice versa. Hence*,*

$$H(x_s \ I \ y_t) = 0 < H(x_s \ I \ y_s) = H(x_s) = 1. \tag{1}$$

thus, information routes from x to y.

Definition 2*:* an implicit routing of information occurs when information flows from x to y without an explicit assignment of the form $y := f(x)$, where $f(x)$ is an arithmetic expression with the variable x. The routing of information occurs, not because of an assignment value of *x*, but because of a routing control based on the value of *x*. This demonstrates that analyzing programs for assignments to detect information routing is not enough. To detect all routing of information, implicit routing must be examined.

3. Execution Based Mechanism

The goal of an execution-based mechanism is to prevent an information routing that violates policy. Checking the routing requirements of explicit route achieves this result for statements involving explicit routings. Before the assignment $y = f(x_1, ..., x_n)$ is executed, the execution-based mechanism verifies that $lub(\underline{x}_1, ..., \underline{x}_n) \le \underline{y}$ if the condition is true, the assignment proceeds. If not, it fails. A naive approach, then, is to check information routing conditions whenever an explicit routing occurs. Implicit routing complicates checking.

Example: let x and y be variables. The requirement for certification for a particular statement
$y \ op \ x$ *is that* $\underline{x} \le \underline{y}$.

The conditional statement if x = 1 then y := a, Causes a routing from x to y. Now, suppose that when $x \ne 1$, $\underline{x} = High$ *and* $\underline{y} = Low$. If routing were verified only when explicit, and $x \ne 1$, the implicit routing would not be checked. The statement may be incorrectly certified as complying with the information routing policy.

3.1. Variables Classes

The classes of the variables in the examples above are fixed. This suggests a notion of dynamic classes, wherein a variable can change its class. For explicit assignments, the change is straight forward. When the assignment y:= f(x_1,… ,x_n) occurs, y's class is changed to $lub(x_1,...,X_n)$. Again, implicit routing complicates matters.

Example: Consider the following program (which is the same as the program in the example for the data mark machine [8].

```
proc copy (x : integer class { x } ;
var y: integer class { y });
var z : integer class variable {Low };
begin
    y := 0;
    z :=0;
    if x=0 then z := 1;
    if z =0 then y := 1;
end;
```

In this program, $\underline{z}$ is variable and initially *Low*. It changes when something is assigned to z .Routings are certified whenever anything is assigned to y. suppose $\underline{y} < \underline{x}$. If $x = 0$ initially, the first statement checks that $Low \le \underline{y}$ (trivially true). The second statement sets z to 0 and $\underline{z}$ to Low. The third statement changes $\underline{z}$ to1 and $\underline{z}$ to $lub(Low, x) = \underline{x}$. The fourth statement is skipped (because $z = 1$). Hence, y is set to 0 on exit. *If* $x = 1$ initially, the first statement checks that $Low \le \underline{y}$ (trivially true). The second statement sets $\underline{z}$ to 0 and z to *Low*. The third statement is skipped (because $x = 1$). The fourth statement assigns 1 to y and checks that $lub(Low, \underline{z}) = Low \le \underline{y}$ (again, trivially true). Hence, y is set to 1 on exit. Information has therefore routed from x to y even though $\underline{y} < \underline{x}$. The program violates the policy but is nevertheless certified.

4. Compiler Based Mechanism

Compiler-based mechanisms check that information routing throughout a program are authorized. The mechanisms determine if the information routing in a program could violate a given information routing policy. This determination is not precise, in that secure paths of information routing may be marked as violating the policy; but it is secure, in that no unauthorized path along which information routing will be undetected.

Definition 3: a set of statements is certified with respect to an information routing policy if the information routing within that set of statements does not violate the policy.

Example: consider the program statement

```
if x = 1 then y := a
else y := b;
```

By the rules discussed earlier, information routes from x and *a* to y or from *x* and *b* to *y*, so if the policy says that, $\underline{a} \le \underline{y}$ $\underline{b} \le \underline{y}$, and $\underline{x} \le \underline{y}$ then the information routing is secure. But if $\underline{a} \le \underline{y}$ only when some other variable *z = 1*, the compiler-based mechanism must determine whether *z = 1* before certifying the statement. Typically, this is infeasible. Hence, the compiler-based mechanism would not certify the statement. The mechanisms described here follow those developed by denning [4].

4.1. Declarations

For our discussion, we assume that the allowed routing is supplied to the checking mechanisms through some external means, such as from a file. The specifications of allowed routing involve security classes of language constructs. The program involves variables, so some language construct must relate variables to security classes. One way is to assign each variable to exactly one security class. We opt for a more liberal approach, in which the language constructs specify the set of classes from which information may route into the variable. For example x: integer class { A, B } states that *x* is an integer variable and that data from security classes *A* and *B* may route into *x*. Note that the classes are statically, not dynamically, assigned. Viewing the security classes as a lattice, this means that *x*'s class must be at least the least upper bound of classes *A* and *B* that is, $lub\{A, B\} \le \underline{x}$.

Two distinguished classes, Low and High, represent the greatest lower bound and least upper bound, respectively, of the lattice. All constants are of class Low. Information can be passed into or out of a procedure through parameters. We classify parameters as input parameters (through which data is passed into the procedure), output parameters (through which data is passed out of the procedure), and input/output parameters (through which data is passed into and out of the procedure). Consider the following program which is the same as the program in the example in [5].

```
(* input parameters are named $i_s$; output  parameters, $o_s$; *)
(* and input/output parameters, $io_s$, with s a subscript *)
    proc something($i_1$, ..., $i_k$; var $o_1$, ..., $o_m$, $io_1$, ..., $io_n$);
    var $l_1$, ..., $l_j$;  (* local variables *)
    begin
        S;   / * body of procedure *)
    end;
```

The class of an input parameter is simply the class of the actual argument *i_s: type class { i_s }*, let r_1, ..., r_p be the set of input and input/output variables from which information routing to the output variable o_s. The declaration for the type must capture this o_s: *type class* $\{r1, ..., r_p\}$.

We implicitly assume that any output-only parameter is initialized in the procedure. The input/output parameters are like output parameters, except that the initial value (as input) affects the allowed security classes. Again, let $r_1, ..., r_p$ be defined as above. Then io_s: *type class* $\{r_1, ..., r_p, io_1, ..., io_k\}$.

Example: consider the following procedure for adding two numbers.

```
proc sum(x: int class { x };
    var out: int class { x, out });
begin
    out := out + x;
end;
```

Here, we require that $\underline{x} \le \underline{\text{out}}$ and $\underline{\text{out}} \le \underline{\text{out}}$ (the latter holding because $\le$ is reflexive). The declarations presented so far deal only with basic types, such as integers, characters, floating point numbers, and so forth. Nonscalar types, such as arrays, records (structures), and variant records (unions) also contain information. The rules for information routing classes for these data types are built on the scalar types.

Consider the array *a*: array 1 .. 100 of *int*; first, look at information routing out of an element *a[i]* of the array. In this case, information routing from *a[i]* and from *i*, the latter by virtue of the index indicating which element of the array to use. Information routing into *a[i]* affect only the value in *a[i]*, and so do not affect the information in *i*. Thus, for information routing from *a[i]*, the class involved is $lub\{\, \underline{a[i]}, \underline{i}\, \}$; for information routing into *a[i]*, the class involved is $\underline{a[i]}$.

5. Program Statements

A program consists of several types of statements some of them typically are conditional statement, *Goto* statement and procedure calls. We use the same statements for our compiler based approach.

5.1. Conditional Statements

A conditional statement has the form

```
if f(x1, ..., xn) then
    S1;
else
    S2;
end;
```

where $x_1, ..., x_n$ are variables and *f* is some (boolean) function of those variables. Either S_1 or S_2 may be executed, depending on the value of *f*, so both must be secure. As discussed earlier, the selection of either S_1 or S_2 imparts information about the values of the variables $x_1, ..., x_n$, so information must be able to route from those variables to any targets of assignments in S_1 and S_2. This is possible if and only if the lowest class of the targets dominates the highest class of the variables $x_1, ..., x_n$. Thus, the requirements for the information routing to be secure are:

- S_1 secure
- S_2 secure
- lub$\{\underline{x}_1, ..., \underline{x}_n\} \le glb\{\underline{y}$ | y is the target of an assignment in S_1 and S_2 }

As a degenerate case, if statement S_2 is empty, it is trivially secure and has no assignments.

Example: consider the statements

```
if x + y < z then
    a := b;
else
    d := b * c - x;
end;
```

Then the requirements for the information routing to be secure are $\underline{b} \le \underline{a}$ for S_1 and $lub\{\underline{b}, \underline{c}, \underline{x}\} \le$ d for S_2. But the statement that is executed depends on the values of *x*, *y*, and *z*. Hence, information also routes from *x*, *y*, and *z* to *d* and *a*. So, the requirements are $lub\{\underline{y}, \underline{z}\} \le \underline{x}, \underline{b} \le a$, and $lub\{\underline{x}, \underline{y}, \underline{z}\} \le glb\{\underline{a}, \underline{d}\}$.

5.2. Goto Statements

A *goto* statement contains no assignments, so no explicit routing of information occurs. Implicit routing may occur; analysis detects these routing.

Definition 4: a basic block is a sequence of statements in a program that has one entry point and one exit point.

Example: consider the following code fragment from [5] adopted for our method.

```
        proc transmatrix (x: array[1..10] [1..10] of int
                                   class{x})
    var y: array [1..10][1..10] of int class{y}};
    var i , j : int class {tmp}
    begin
      i : =1
      {b1}
      12: if i>10          goto 17
      {b2}
      j=1;
      {b3}
      14: if j>10 then  goto 16;
      {b4}
      y[j][i]= x[i][j];
      {b5}
      j:=j+1;
      goto 14;
      16:i:=i+1;
      {b6}
      goto 12;
      17:
      {b7}
  end;
```

There are seven basic blocks, labeled b_1 through b_7 and separated by lines. The second and fourth blocks gave two ways to arrive at the entry either from a jump to the label or from the previous line. They also have two ways to exit either by the branch or by falling through to the next line. The 5th block has three lines and always ends with a branch. The sixth block has two lines and can be entered either from a jump to the label or from the previous line. The last block is always entered by a jump.

Control within a block routing from the first line to the last. Analyzing the routing of control within a program is therefore equivalent to analyzing the routing of control among the program's basic blocks. Figure 1 shows the routing of control among the basic blocks of the body of the procedure Transmatrix.

The basic blocks are labeled b_1 through b_7.The conditions under which branches are taken are shown over the edges corresponding to the branches.

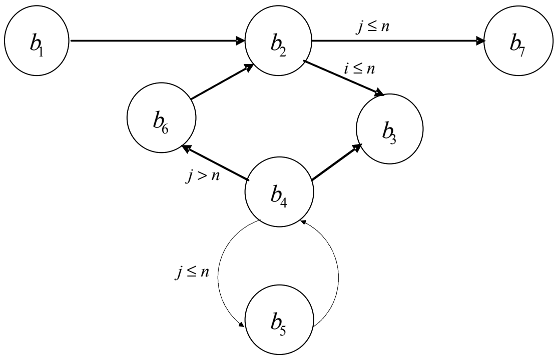


Figure 1. The control routing graph of the procedure transmatrix.

When a basic block has two exit paths, the block reveals information implicitly by the path along which control routing. When these paths converge later in the program, the (implicit) information routing derived from the exit path from the basic block becomes either explicit (through an assignment) or irrelevant. Hence, the class of the expression that causes a particular execution path to be selected affects the required classes of the blocks along the path up to the block at which the divergent paths converge.

Definition 5: an immediate forward dominator of a basic block b (written IFD(b)) is the first block that lies on all paths of execution that pass through b for example in the procedure transmatrix, the immediate forward dominators of each block are $iFD(b_1) = b_2, IFD(b_2) = b_7, IFD(b_3) = b_4 = b_6 IFD(b_5) = b_4$, and $IFD(b_6) = b_2$. Computing the information routing requirement for the set of blocks along the path is now simply applying the logic for the conditional statement. Each block along the path is taken because of the value of an expression. Information routing from the variables of the expression into the set of variables assigned in the blocks. Let B_i be the set of blocks along an execution path from b$_1$ to IFD(b$_i$), but excluding these endpoints. Let $X_{i1}, \cdots X_{in}$ be the set of variables in the expression that selects the execution path containing the blocks in B_j. The requirements for the program's information routing to be secure are: all statements in each basic block secure $lub\{\underline{x_{i1}}, \ldots \underline{x_{in}}\}$ {y | y is the target of an assignment in B_i }.

Example: consider the body of the procedure transmatrix. We first state requirements for information routing within each basic block:

$b_1 : low \le \underline{i} \Rightarrow secure$

$b_1 : low \le \underline{j} \Rightarrow secure$

$b_5 : lub\,\{\underline{x[i][j]}, \underline{i}, \underline{j}\} \le \underline{y} \le \underline{y[j][i]}; \le \underline{j} \le \underline{j}$

$\Rightarrow lub\{\underline{x[i][j]}, \underline{i}, \underline{j}\} \le \underline{y[j][i]}$

$b_6 : lub\,\{low, \underline{i}\} \le \underline{i} \Rightarrow secure$

The requirement for the statements in each basic block to be secure is:

$for\, i = 1, \cdots, n\ and\ \ j = 1, \cdots n, \mathrm{lub}\ \{\underline{X[i][j]}, \underline{i}, \underline{j}\} \ \le\ \underline{y[j][i]}$

.

By the declarations: this is true when $\mathrm{lub}\{\underline{X}, \underline{i}\} \ \le\ \underline{y}$. In this procedure, $B_2 = \{b_3, b_4, b_5, b_6\}\ and\ B_4 = \{b_5\}$. Thus, in B_2, statements assign values to $i, j,\ and\ y[j][i]$. In B_4, statements assign values to $j\ and\ y[j][i]$. The expression controlling which basic blocks in B_2 are executed is $i \le 10$; the expression controlling which basic blocks in B_4, are executed is $j \le 10$. Secure information routing requires that i ≤ glb {$\underline{i}$, $\underline{y}$} and i ≤ glb {$\underline{i}$, $\underline{y}$}, or $\underline{i} \le \underline{y}$ combining these requirements, the requirement for the body of the procedure to be secure with respect to information routing is $\mathrm{lub}\left\{\underline{X}, \underline{i}\right\} \le \underline{Y}$.

5.3. Procedure Calls

A procedure call has the form

proc procname(i$_1$, ..., i$_m$: int; var o$_1$, ..., o$_n$: int);
begin
S;
end;

where each of the i_j's is an input parameter and each of the o_j's is an input/output parameter. The information routing in the body *S* must be secure. As discussed earlier, information routing relationships may also exist between the input parameters and the output parameters. If so, these relationships are necessary for *S* to be secure. The actual parameters (those variables supplied in the call to the procedure) must also satisfy these relationships for the call to be secure. Let x_1, ..., x_m and y_1, ..., y_n be the actual input and input/output parameters, respectively. The requirements for the information routing to be secure are *S* secure

For $j = 1, \ldots, m$ *and* $k = 1, \ldots, n$, *if* $i_j \leq o_k$ *then* $x_j \leq y_k$
For $j = 1, \ldots, n$ *and* $k = 1, \ldots, n$, *if* $o_j \leq o_k$ *then* $y_j \leq y_k$

Example: consider the procedure transmatrix from section 5.2. As we showed there, the body of the procedure is secure with respect to information routing when $lub\{\underline{x}, \underline{tmp}\} \leq y$. This indicates that the formal parameters x and y have the information routing relationship $\underline{x} \leq \underline{y}$. Now, suppose a program contains the call transmatrix (a, b). The second condition asserts that this call is secure with respect to information routing if and only if $\underline{a}$.

6. Conclusion

This paper focus on the language based information routing security. Two mechanisms are put forward complier based and execution based mechanism to specify and enforce security policies with C++ language. We have demonstrated that it is possible to implement security policy using security-typed languages through examples and C typed codes. However, further investigation of the language based support for policy enforcement is necessary before they can fulfill their considerable promise of enabling more secure routing. For example certifying compilers are needed for security-typed languages, because compilers for source languages (such as Jif) are too complex to be part of the trusted computing base. However, current security-type systems are not expressive enough to support a security-typed low-level target language.

From our finding the results demonstrate that the idea of language based is easy to comprehend but much more difficult to implement efficiently. Here are some obstacles that we have learned from the paper:

- How to encode the formal language? Trivial encoding of policy properties programs is very large.
- How to check the policy? This is not an easy task if you want your policy to be terse and the checker to be small, fast, and mostly-independent of the actual safety policy that is being enforced.
- How to relate the policy with the program? It is of no use to validate the policy if we cannot ensure that it says something about the program at hand.

Although many type secure languages exists fundamentally our (C++) based codes makes it impossible to commit broad classes of errors to policy enforcement. Our work in C++ also uncovered three central deficiencies. First aspects of information and enforce security policies with C++ language. We have demonstrated that it is possible to implement security policy using security-typed languages through examples. Given the value of one variable, entropy measures the amount of information that one can deduce about a second variable. Second the routing can be explicit, as in the assignment of the value of one variable to another, or implicit, as in the antecedent of a conditional statement depending on the conditional expression. Third traditionally, models of information routing policies form lattices. Should the models not form lattices, they can be embedded in lattice structures. Hence, analysis of information routing assumes a lattice model.

Our approach through examples and definitions addresses incongruity insecurity by allowing flexibility to environment applied as security requirements are as diverse as the environments in which systems exist, support for flexible policy- defined security is desirable. Definitions presented elaborate how command sequence in C language causes information routing from point *x* to *y*. However conditional statements has been used, (boolean) and variables function for execution

Even in the face of the considerable challenges we encountered in this project, we are heartened by the experience. To be sure, the tools and practice of using C++ codes, and in larger sense security-typed languages, must mature before their promise is met. We see this work as another step in that maturation and take this work as another milestone in that achievement.

References

[1] Askarov L. and Sabelfeld A., "Secure Implementation of Cryptographic Protocols: A Case Study of Mutual Distrust," *in Proceedings of the 10th European Symposium on Research in Computer Security ESORICS '05*, pp. 1-5, Italy, 2005.

[2] Bell D. and La Padula L., Secure Computer Systems, *Mathematical Foundations Technical Report*, 1973.

[3] Chong S. and Myers A., "Decentralized Robustness," *in Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pp. 321-334, USA, 2006.

[4] Denning D., *Cryptography and Data Security, Reading*, MA, 1982.

[5] Goguen J. and Meseguer J., "Security Policies and Security Models," *in Proceedings of IEEE Symposium on Security and Privacy,* pp. 11-20, USA, 1982.

[6] Hicks B., King D., McDaniel P., and Hicks M., "Trusted Declassification: High-Level Policy for a Security-Typed Language," *in Proceedings of Workshop on Programming Languages and Analysis for Security*, pp. 65-74, Canada, 2006.

[7] Kent S. and Atkinson R., "Security Architecture for the Internet Protocol," *Internet Engineering Task Force Journal*, vol. 37, no. 1, pp. 1, 1998.

[8] Mantel H. and Sabelfeld A., "A Unifying Approach to the Security of Distributed and Multi Threaded

Programs," *Journal of Computer Security*, vol. 11, no. 4, pp. 615-676, 2003.

[9] Myers C., "Mostly-static Decentralized Information Flow Control," *Technical Report MIT/LCS/TR-783*, 1999.

[10] Myers C., Nystrom N., Zheng L., and Zdancewic S., "Jif: Java + Information Flow," www.cs.cornell.edu/jif, July 2001.

[11] Montgomery D. and Murphy S., "Towards Secure Routing Infrastructures," *IEEE Security & Privacy,* vol. 4, no. 5, pp 84-87, 2006.

[12] Pottier F. and Simonet V., "Information Flow Inference for ML," *in Proceedings of Principles of Programming Languages (POPL)*, pp. 319-330, USA, 2002.

[13] Sabelfeld A. and Myers A., "Language Based Information Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5-19, 2003.

[14] Simonet V., "FlowCaml in a Nutshell in Hutton," *in Proceedings of the First APPSEM-II Workshop*, pp. 152-165, UK, 2003.

[15] The Internet Engineering Task Force, www.ietf. org/html.charters/rpsec-charter.httm, 2006

[16] Volpano D. and Smith G., "Probabilistic Noninterference in a Concurrent Language," *Journal of Computer Security*, vol. 7, no. 2, pp. 231-253, 1999.

[17] Volpano D., Smith G., and Irvine C., "A Sound Type System for Secure Flow Analysis," *Journal of Computer Security*, vol. 4, no. 3, pp. 167-187, 1996.

[18] Ylonen T., "SSH: Secure Login Connections Over the Internet," *in Proceedings of 6th USENIX UNIX Security Symposium*, pp. 37-42, Korea, 1996.

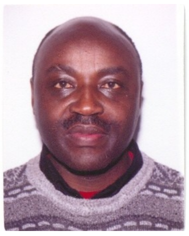
[19] Zdancewic S., "A Type System for Robust Declassification," *in Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics*, pp. 47-66, Berlin, 2003.

George Oreku received his Master in computer science from University of Odessa Polytechnic in 2002. He is currently a PhD candidate at the Department of Computer Science and Engineering, Harbin Institute of Technology, Harbin, China.

Li Jianzhong the director of the Department of Computer Science and Engineering at the Harbin

Institute of technology, China. Also he is a part-time professor in FuDan University and RenMin University of China.

Fredrick Mtenzi is a supervisor of postgraduate students, lecturing systems security and cryptography, security and forensics, security and



cryptography advanced research, and proposal writing at school of Computing Dublin Institute of Technology, Ireland.