# Information Flow Control in C using the Decentralized Label Model *

Andrew Habib

Department of Applied Mathematics and Computer Science,
Technical University of Denmark
Richard Petersens Plads
Building 324, 2800 Kgs. Lyngby, Denmark
s135552@stud.dtu.dk

**Abstract.** To date, C programing language still dominates sequential programing and many applications from embedded systems to large-scale complex systems use C in a way or another. Hence was the need to develop techniques to ensure confidentiality and integrity of data manipulated by applications written in C. Through this work, we investigate the idea of applying the information flow control model called the *Decentralized Label Model* to the C programming language.

**Keywords:** Information flow, declassification, decentralized label, C

## 1 Introduction

IT security is of increasing importance nowadays due to the huge move and dependency on software tools and applications in all aspects of life. Security requirements mainly focus on two main goals: protecting confidentiality and integrity of data. The traditional approach to both problems has been to use access control techniques to regulate who can access sensitive data and who is allowed to modify it. But using access control methods cannot solve the problem fully. If we consider the data confidentiality problem, it is not sufficient to manage who or which application has access to the private data; because once a legal user or program gains access to it, there is no way to control how the data might be distributed and released to other probably unauthorized viewers. Therefore was the need to develop the concept of end-to-end security where information security is ensured even when untrusted users or applications have to manipulate or deal with the sensitive information.

To achieve this goal of finer-grained control over information dissemination and propagation, information flow control models have been proposed. To mention some, the work in [4] and by [9] represent early ideas in this area. But the problem with these models is that they are either too restrictive and they incur huge run-time overhead or they are not practically applicable.

* This report is submitted in fulfillment of an independent guided research course titled "02295 Advanced Topics in Computer Science" for 5 ECT as part of the authors masters degree.

Late ninetieth, a new model of information flow control, named the *decentralized label model* (called DLM from this point on) has been proposed by Andrew C. Myers and Barbara Liskov to manage information flow in an environment with mutual distrust and decentralized authority [6]. The two main goals of this model are to solve the problem of unduly strict old models and to be more practical and hence applicable and usable in real systems. Recently, the DLM has been successfully applied to Java and resulted in the Jif (Java information flow) programming language [7].

The purpose of this work is to explore the idea of using the DLM in the C programming language. As C language is one of the most used programing languages of all times [11, 12], interest in securing applications written fully or partially in C never fades.

The rest of this paper is organized as follows. Section 2 presents the DLM in details. Then in section 3 we discuss potential problems arising when adapting the DLM to C language. Section 4 shows that if we restrict our approach to a subset of C that avoids the potential problems, then we can safely apply the DLM to C. In section 5 we describe our proposal to solve one of the problems we identified earlier. Section 6 concludes the paper and section 7 discusses future work.

## 2 The Decentralized Label Model (DLM)

### 2.1 Abstract Model

The *decentralized label model* as the name suggests, is an information flow control approach that is intended for computing environments with decentralized authority and mutual distrust. The model allows users and applications to fully control the propagation and dissemination of information they own with no need to delegate their trust to a central authority. To provide finer-grained and precise control over information distribution with minimal run-time overhead, the model is designed as a static analysis technique applied at the programing language level (i.e source code).

The main idea is to label data with its owner(s) and to whoever an owner is willing to release this data. The authorized viewer(s) of the data - and potentially owner(s) themselves - are added and removed solely at the discretion of the current data owner(s) according to strict rules, called relabeling rules. These labels are applied and checked (mostly) statically at compile time so as to ensure that the program does not violate the intended information flow policy at runtime. In the following subsections we summarize the important components of DLM and how they work.

#### 2.1.1 Principals and Labels

The DLM consists of two main components: principals and labels. A principal is a user or some authority entity which could be a role or group of users for example. The set of principals is organized in a hierarchy defined by the *acts*

*for* relation. This relation is reflexive and transitive and hence gives a partial ordering on the set of principals. At any point, a principal $P_1$ could check whether it is allowed to act on behalf of another principal $P_2$ by checking the function $actsfor(P_1, P_2)$. Inside a system, a process could be allowed to run on behalf of a principal or a set of principals if the principal running the process can act for the designated principal.

Labels are treated as types and hence values, slots (source or destination variables, objects, ...) and (input and output) channels all receive labels. A label $L$ is composed of a set of owners: $owners(L)$ and for each owner $O$, the label contains a set of associated readers: $readers(L, O)$. The $owners(L)$ is the set of principals owning the data and therefore, are the ones in control of its dissemination. Each owner controls his set of readers $readers(L, O)$ which represents the set of principals the owner is willing to release the data to. A set of *effective readers* of a label $L$ is the intersection of all readers sets which gives the set of readers to which all owners mutually agree to release the data. For example, a label could be of the form $L = \{o1 : r1; o2 : r1, r2, r3\}$ where $owners(L) = \{o1, o2\}$, $readers(L, o1) = \{r1\}$ and *effective readers* of $L$ is $\{r1\}$. From this point on, the label of an expression $e$ is denoted by $\underline{e}$.

### 2.1.2 Relabeling Rules

The model defines two main relabeling rules:

1. **Restriction**

   Relabeling from $L_1$ to $L_2$ is called a restriction when it either adds owners, removes reader or both. $L_1 \sqsubseteq L_2$ denotes a restriction from $L_1$ to $L_2$ where

   $$L_1 \sqsubseteq L_2 \Leftrightarrow \begin{array}{r} owners(L_1) \subseteq owners(L_2) \\ \forall O \in owners(L_1) : readers(L_1, O) \supseteq readers(L_2, O) \end{array}$$

   Intuitively, if we remove reader(s) allowed by an owner in $L_1$, the resulting label $L_2$ is more restrictive since fewer readers are permitted. But why adding owners is also a restriction? This is because a new owner $O$ comes with his set of allowed readers $readers(L, O)$ and eventually this could restrict the *effective reader* of $L$ as we discussed earlier. Therefore, restriction is always a safe operation with respect to data confidentiality and hence it requires no special privilege to be preformed at any point.

2. **Declassification**

   A declassification either adds readers for an owner $O$ or removes the owner $O$ itself. It is permitted only when the process doing it is acting for $O$. Declassification strictly depends on the principal hierarchy given by the *acts for* relation. One can see that the principal hierarchy is very crucial to the model and hence it should be carefully decided according to the precise need of a system.

Secure information flow is achieved by enforcing that relabeling a value is permitted only when the new label is consistent with the original flow policy

(i.e. label). Therefore, a convention assumed by the model is that labels on values do not change; but rather whenever a value is relabeled, a new copy of the value is created with the new label.

#### 2.1.3 Joining Labels

In a program, values are computed by combining other values. So the model defines the label of a derived value as follows. The label of a value resulting from combining two values with labels $L_1$ and $L_2$ is given by a join operation on the two labels: $L_1 \sqcup L_2$ where

$$owners(L_1 \sqcup L_2) = owners(L_2) \cup owners(L_2)$$
$$readers(L_1 \sqcup L_2, O) = readers(L_1, O) \cap readers(L_2, O)$$

$L_1 \sqcup L_2$ is the least restrictive label of both and is obtained by taking the union of the two owners sets and the intersection of each corresponding readers sets. This definition is concise if we let

$$O \notin owners(L) \Rightarrow readers(L, O) = \textit{set of all principals}$$

#### 2.1.4 Lattice of Labels

The restriction operator $\sqsubseteq$ forms a partial ordering on the set of labels. Hence, such set of labels forms a lattice with:

- Partial order: the restriction operator $\sqsubseteq$
- Least element (bottom) $\bot$ which represents the least restrictive label data can ever have, implying data with no restriction at all. And we can define: $owners(\bot) = \phi$ and $readers(\bot, O) = \textit{set of all principals}$.
- Greatest element (top) $\top$ which represents the most restrictive label data can ever have, implying data that cannot flow any where. And we can define: $owners(\top) = \textit{set of all principals}$ and $readers(\top, O) = \phi$.
- Least upper bound (join) $\sqcup$, which is the join operator we defined earlier.
- Greatest lower bound (meet) $\sqcap$, which is the dual of the join operator (i.e. the intersection of owners and union of corresponding readers sets.

### 2.2 Model Applied to a Simple Language

In the following sections, we briefly describe how the DLM is applied to a programing language that supports basic operations like assignments, conditional branches and procedures. For more details, we refer the reader to [6].

#### 2.2.1 Additions to Conventional Language

The programing language needs to be augmented with the following:

- All variables, procedures arguments and return values have labeled types which are normal types (like `int`) and a static label expression.

- There is an `if_acts_for(`$P_1, P_2$`)` $S_1$ `[else` $S_2$`]` statement which tests whether principal $P_1$ is allowed to act on behalf of principal $P_2$. The *effective authority* of a principal (or piece of code) is the set of principals the code can act for. If the check is successful, then the *effective authority* of $P_1$ is increased to include $P_2$, and $S_1$ gets executed. If the test fails, optionally $S_2$ is executed, if any.
- There is a `declassify(e,`$L$`)` operator that declassifies an expression `e` to label $L$. This operation is permitted only inside an `if_acts_for` statement. Declassification of `e` to $L$ is safe if $\underline{\texttt{e}} \sqsubseteq L \sqcup L_A$ where $L_A$ is a special label for which $owners(L_A)$ is the set of all principals in the *effective authority* and $\forall O \in owners(L_A) : readers(L_A, O) = \phi$.
- A calling procedure can grant part or all of its authority to the callee, and the callee can test for this authority using the `if_acts_for` statement.

### 2.2.2 Implicit Flow and Block Label

Control flow can implicitly leak information about sensitive data when flow branches on confidential data and then computations are performed on publicly observed slots. For example, `x=0; if (y>0) x=1;` leaks information about `y` by observing the value of `x`. That is why for an assignment `x=e` to be safe, it is not sufficient to enforce that $\underline{\texttt{e}} \sqsubseteq \underline{\texttt{x}}$.

To solve this problem, the model uses a block label. All values $v_i$ that are observed in order for a program execution to reach a specific point are joined together using the predefined join operator $\sqcup$ to form the block label: $\underline{B} = \bigsqcup_i \underline{v_i}$. This block label $\underline{B}$ indicates information that could be inferred by realizing that the program execution reaches the specific basic block $B$. Such block label is easily computed statically from the basic block diagram of a program.

### 2.2.3 Assignment Statements

Using the block label idea explained above, an assignment statement `x=e` inside a block $B$ is safe if and only if $\underline{\texttt{e}} \sqcup \underline{B} \sqsubseteq \underline{\texttt{x}}$.

### 2.2.4 Conditional Statements

The conditional statements `if` and `while` are similar to each other. It follows from the discussion above regarding implicit information flow that a statement of the form `"if (b) S"` or the statement `"while (b) S"` are legal if $S$ is safe under the label $\underline{\texttt{b}} \sqcup \underline{B}$.

### 2.2.5 Explicit Values

Any expression containing only explicit data is labeled with $\bot$ since the expression itself does not leak information about any other piece of data. For example, the statement `x=10;` has label constraint $\bot \sqsubseteq \underline{\texttt{x}}$ and surely it is safe.

#### 2.2.6 Label Polymorphism

For code reusability sake, the idea of implicit labels to procedures is introduced. When calling a function that takes a parameter, if the label of that parameter is not explicitly specified, then it is inferred implicitly from the label of the actual argument. This provides us with label polymorphism and hence allows functions to be generic with respect to parameters labels. Therefore, only a single version of the function needs to be defined.

#### 2.2.7 Solving the System of Constraints

Label constraints generated for each basic block are combined all together to form a system of constraints to be solved. A solution to the system implies adherence to the information flow policy and hence the code is secure. On the other hand, inconsistency that result in a contradiction inside the constraint system implies a potential information leak. We omit the algorithm for solving the system of constraints from this paper and we refer the reader to [6] for more details.

At this point, we have introduced all important parts of the DLM and explained how it could be applied to a programing language.

## 3 DLM and C

Now we analyse how feasible it is to apply the DLM to the C programing language. We identify major C artifacts that are hard to protect using the DLM.

### 3.1 C Pointers

C language is very famous for its highly versatile but problematic pointer variables and their associated operators. A pointer is a special variable that holds the memory address of (points to) another variable and a pointer variable should have the same type of the variable it points to. Pointer variables support assignment, arithmetic (except for division and multiplication) as well as relational operators in the usual sense. Now, if it was only this and ignoring any possible run-time memory corruption resulting from manipulating memory addresses and their content, then it is straight forward to apply the DLM to pointers like any other slots in the code and the usual rules explained above should apply accordingly. Unfortunately, it is not.

C provides two operators to make the best use of pointer variables: the dereference operator $*$ and the address-of operator `&`. The first is used to directly access (dereference) the actual variable pointed to by a pointer and the second returns the memory address of a variable which could be assigned to a pointer variable for later use. These operators are very powerful because they provide extra information about variables and more capabilities to deal with slots and values in a program.

Yet, these handy functionalities come at a high cost. The extras they offer pose a serious threat to information flow control as they carry sensitive pieces of information about variables and slots; not to mention their unpredictable run-time behaviour which also could compromise an information flow policy. Therefore, C pointers should receive special consideration in the context of protecting the confidentiality of data processed by a program.

### 3.2 Weakly-typed C

Since [static] information flow checks are pursued as extensions to [static] type checking [6], the decentralized label model is applicable to strongly, statically typed languages [7]. And that is why Java was the natural choice for applying the DLM model, according to the model authors. But this is another problem when we try to apply the DLM to C as C is a weakly - but statically - typed language.

Weak typing in C is present in the form of: implicit type conversions (for example: between **`char`** and **`int`**), untyped (**`void`**) pointers,pointers exposition (pointer arithmetic operations can bypass C type system) and untagged unions (where a single value of a specific type could be viewed as if it is of another type). Such unsafe operations hinder static analysis techniques unuseful because they can lead to unexpected run-time memory corruption in ways unpredictable by the compiler at static-time [3]. Moreover, run-time errors and failures due to weak typing of C can leak information (covert channels [5]).

### 3.3 Other C Functionalities

Additionally, C provides unusual capabilities that are very desirable for many purposes, but are hard to accommodate for, when trying to apply the DLM. Inline assembly code is a special property of C which is intended for taking advantage of the speed of machine language. But it makes information flow tracking very complicated. Think of it as computations are done on raw data (bit level) irrespective of any type system constraints and with much less control over program flow. Therefore, tracking information flow at this low level, is very sophisticated due to the absence of high level abstractions and increased flexibility offered by assembly languages [10].

Also pointers to functions complicate tracking the program flow in static time and could result in information leakage as well.

### 3.4 Can we still apply the DLM to C?

Though it is arguable that some of these problems could be solved by doing run-time checks, we should keep in mind that one of the main advantages of the DLM model is that it is a static analysis technique. If we opt to perform run-time validations to prohibit information leaks incurred by these functionalities, then

we run into the drawbacks of dynamic information flow control: the huge runtime overhead, covert channels and the difficulty of detecting implicit information flow. And this renders the DLM unuseful.

Moreover, in addition to being extremely expensive and infeasible, models that detect unsafe memory operations for weakly typed languages are heuristic and provide no guarantee that all such errors will be detected [3].

That said, in the next section we show that if we restrict our attention to a subset of C that avoids all of the problematic structures we identified here, then the DLM could be applied to this restricted subset of C.

## 4 DLM with Subset of C

We show that the DLM concepts of the principal hierarchy, labels and relabeling rules are applicable to a subset of the C language - in a similar fashion to Jif - where we omit the use of C problematic structs that we identified above. We do so through a small example of a function written in C and show how the rules could be applied.

`decrypt` is a simple decryption function of the famous RSA public-key cryptosystem. It takes as input the cipher text `C`, the RSA modulus `n` and the private-key (decryption exponent) `d` and returns the decrypted message `Msg` or `0` otherwise. We present one explicit principal in the usage scenario: `rol` which represents the role that runs the procedure itself.

Labels of the ciphered message $\underline{\texttt{C}}$ and the decryption exponent $\underline{\texttt{d}}$ are both implicit parameters to the function and hence the function is polymorphic and could be used to perform decryption of messages with different security classes using private-keys with diverse ownership requirements. Moreover, the return value of the function - which is the deciphered message - is obtained by doing computations on these values $\underline{\texttt{C}}$, $\underline{\texttt{d}}$, and $\underline{\texttt{n}}$. Therefore, the return value should have label as the join of the two labels $\underline{\texttt{C}} \sqcup \underline{\texttt{d}}$ (we ignored the security class $\underline{\texttt{n}}$ as it is the public RSA modulus where $\underline{\texttt{n}} = \bot$).

```
1 int{C ⊔ d} decrypt(C,d,n){
2     int Msg=1;                    // ⊥ ⊑ Msg
3     for(i=0, i<d, i++)            // L1 = i ⊔ d
4         Msg=Msg*C%n;              // Msg ⊔ C ⊔ n ⊔ {rol} ⊔ L1 ⊑ Msg
5     Msg=Msg%n;                    // Msg ⊔ n ⊑ Msg
6     if_acts_for(decrypt,rol)
7         return declassify(Msg); // Msg ⊑ Ld ⊔ {rol}, Ld ⊑ C ⊔ d
8     return 0;                     // ⊥ ⊑ C ⊔ d
9 }
```

Example: `decrypt` function with generated constraint

Comments next to each line of code show the label constraint for the statement. At line 3, we see the block label $\texttt{L}_1$ of the **`for`** statement. At line 7, the

declassification operator uses a temp variable `L_d` to denote the label of the declassified expression and the **`return`** statement incurs the second constraint on `L_d`.

Solving the generated system of constraint yields the following:

$$\begin{aligned} L_1, \underline{Msg}, \underline{i} &= \underline{C} \sqcup \underline{d} \sqcup \{rol : \phi\} \\ L_d &= \underline{C} \sqcup \underline{d} \end{aligned}$$

## 5 DLM and C Pointers

Now we give insights on how we might solve some aspects of the one problem of C pointers when applying the DLM.

To begin tackling the problem, one can see that we need to separate between a variable and its address. In many usage scenarios, the two could be of different security classes permitting different operations according to the given clearance per principle. One might need to allow a procedure to perform some operation on a set of pointers while enforcing that the procedure is not authorized to access (deference) the actual variables pointed to by the pointers. For example, a free memory procedure that releases memory allocated to pointers should not be allowed to access data pointed-to by the pointer. Therefore, according to this observation, we can give our first modification.

1. **Variable label and address label**
   The variable label $L$ is extended to become $L = \{L_a, L_b\}$ where $L_a$ is the normal label of the variable value and $L_b$ is the label of the address of the variable. For a variable **`int`**$\{L_a, L_b\}$`x`, any computation involving the value of `x` incurs an $L_a$ constraint and computations on the address of the variable (`&x`) enforces $L_b$ constraint. And we can apply the normal relabeling rules. For instance, `y=&x` is safe if it could be determined that $L_{x,b} \sqsubseteq L_{y,a}$. We denote the label of `x` as $\underline{x}_a$ and the label of `&x` as $\underline{x}_b$

Moreover, the labels $L_a$ and $L_b$ are related in the following manner.

$$owners(L_a) \subseteq owners(L_b)$$

This guarantees that if a principal is in the owners set of a specific value, it surely is in the owner set of the address of that value. But the converse is not necessarily true. Additionally, for $o \in owners(L_a)$ we opt not to enforce any relation between $readers(L_a, o)$ and $readers(L_b, o)$ to allow for a flexible yet secure information flow.

In fact, such rule accommodates for various usage scenarios. For example, a value owner has discretion over the value address (in accordance with the entire security policy). On the other hand, an address owner does not necessarily own the value itself and hence might not be able to even read it (i.e. deference a pointer to that value). Also, a value reader might not be able to read the address of this specific value and an address reader may not be allowed to read the value itself too.

But with the dereference operator, things get more complicated. The actual variable a pointer pointing to, might not be known at compile time, and hence evaluating a label constraint will not be possible. For example, we cannot evaluate the label constraint of the assignment `*x=y` because the actual variable pointed to by the pointer `x` is statically unknown, if any.

To solve such a problem, we propose using alias analysis results. Alias analysis (sometimes referred to as points-to analysis and we use the two terms interchangeably) is a static analysis technique used to determine - at compile time - which variable(s) a pointer may (or must) refer to at rum-time. The result of this analysis could be represented in different ways, for example, for each pointer, a set of possible variables referred to by that pointer is given.

We can incorporate the result of an alias analysis into the DLM model so as to predict the possible values a pointer may refer to and then enforce some strict rules to ensure that no information leak could happen.

And we are ready to present the second suggestion.

**2**. **Pointer dereference: which variable?**

Whenever a variable is presented in an expression through a pointer dereference, the label of that pointer dereference expression must be the least restrictive label that respects all restrictions incurred by all possible variables to which the pointer might be pointing. This means that for the assignment `y=*x`, the label constraint is $\bigsqcup_i \underline{x_i}_a \sqsubseteq \underline{y}_a$ where $x_i$ is all variables that the pointer `x` could possibly point to, and are obtained from its alias analysis.

Rule **2**. ensures that a variable of higher security class does not end up in a storage of lower security class through the use of pointer dereference operator. But it is not sufficient on its own since - according to rule **1**. - as we need to differentiate between the variable and its address according to their respective labels. So we should to able to check whether the principle doing the pointer deference operation is allowed access to the actual variable pointed to by the pointer or not. This gives us the following rule.

**3**. **Pointer dereference: who can do it?**

A pointer dereference operation is allowed if and only if the effective authority of the program segment doing the dereference operation includes all possible owners of all possible variables to which the pointer might be pointing to. That is, for all variables $x_i$ in the set of aliases of x:

$$\textit{effective authority} \supseteq \bigcup_i owners(\underline{x_i}_a)$$

The suggested modification in **1**. and the two rules for pointer dereferencing in **2**. and **3**. serve as good ground for applying the DLM to C pointers while enforcing protection against potential information leakage. But it is worth noting that rules **2**. and **3**. depend on the alias analysis of the C source code. However, at run-time, weakly-typed C can easily invalidate the results of points-to analysis due to illegal memory operations [3]. We address this problem in the future work section.

## 6 Conclusion

In this paper, we presented the decentralized label model (DLM) for static information flow control. Then we showed how it could be applied to a programing language to statically validate an application source code against a defined information flow security policy. After that, we discussed potential problems that evolve when we try to adapt the DLM to C programing language.

Finally, we proposed a partial solution to the problem of C pointers and DLM. We suggested that a variable label type should carry two different labels: one for the variable itself (value) and one for the variable address. And we introduced two new rules for checking the pointer dereference expressions. Also, we introduced the idea of incorporating the results of alias analysis into the DLM model to solve the problem of which variable a pointer could actually be pointing-to at run-time.

To sum up, adapting the DLM to C is not straight forward. C has many special properties and characteristics that require special consideration and careful handling when applying the DLM to it. The following section of future work identifies numerous potential directions for further research.

## 7 Future Work

This work is an investigation of the idea of applying DLM to C. Therefore, all the problems we identified in section 3 serve as good directions for future work and research. For example, one could explore the idea of enforcing type-safety on C inline assembly through approaches similar to [1, 2, 10]

One could also investigate if some components of the DLM could be postponed to later stages or even omitted when adapting the DLM to C.

In addition, the problem we identified at the end of section 5 that relates to invalidated points-to analysis due to C weak-typing is worth looking at more thoroughly. This problem, though difficult to solve, has been the target of various research and some promising outcomes are already there. More specifically, one could investigate using tools such as *CCured* [8] or the compilation strategy named *SAFECode* [3] to ensure type-safety and validity of alias analysis during run-time.

## 8 Acknowledgement

The author would like to acknowledge the helpful insights and guidance from professor Flemming Nielson, especially for the approach of tackling the problem.

## References

1. Bonelli, Eduardo, Adriana Compagnoni, and Ricardo Medel. Information flow analysis for a typed assembly language with polymorphic stacks. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Springer Berlin Heidelberg, 2006. 37-56.

2. De Francesco, Nicoletta, and Luca Martini. Instruction-level security analysis for information flow in stack-based assembly languages. Information and Computation 205.9 (2007): 1334-1370.
3. Dhurjati, Dinakar, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. ACM SIGPLAN Notices 41.6 (2006): 144-157.
4. Gray III, James W. Toward a mathematical foundation for information flow security. Journal of Computer Security 1.3 (1992): 255-294.
5. Millen, Jonathan K. Covert channel capacity. 2012 IEEE Symposium on Security and Privacy. IEEE Computer Society, 1987.
6. Myers, A. C. and Liskov, B.: A decentralized model for information flow control. SIGOPS Oper. Syst. Rev. 31, 5 (October 1997), 129-142.
7. Myers, A. C. and Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. 9, 4 (October 2000), 410-442.
8. Necula, George C., Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. ACM SIGPLAN Notices 37.1 (2002): 128-139.
9. Wittbold, J. Todd, and Dale M. Johnson. Information flow in nondeterministic systems. 2012 IEEE Symposium on Security and Privacy. IEEE Computer Society, 1990.
10. Yu, Dachuan, and Nayeem Islam. A typed assembly language for confidentiality. Programming Languages and Systems. Springer Berlin Heidelberg, 2006. 162-179.
11. Programming Language Popularity. `http://www.langpop.com/`. (Retrieved December 2014).
12. TIOBE Programming Community Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. (Retrieved December 2014).