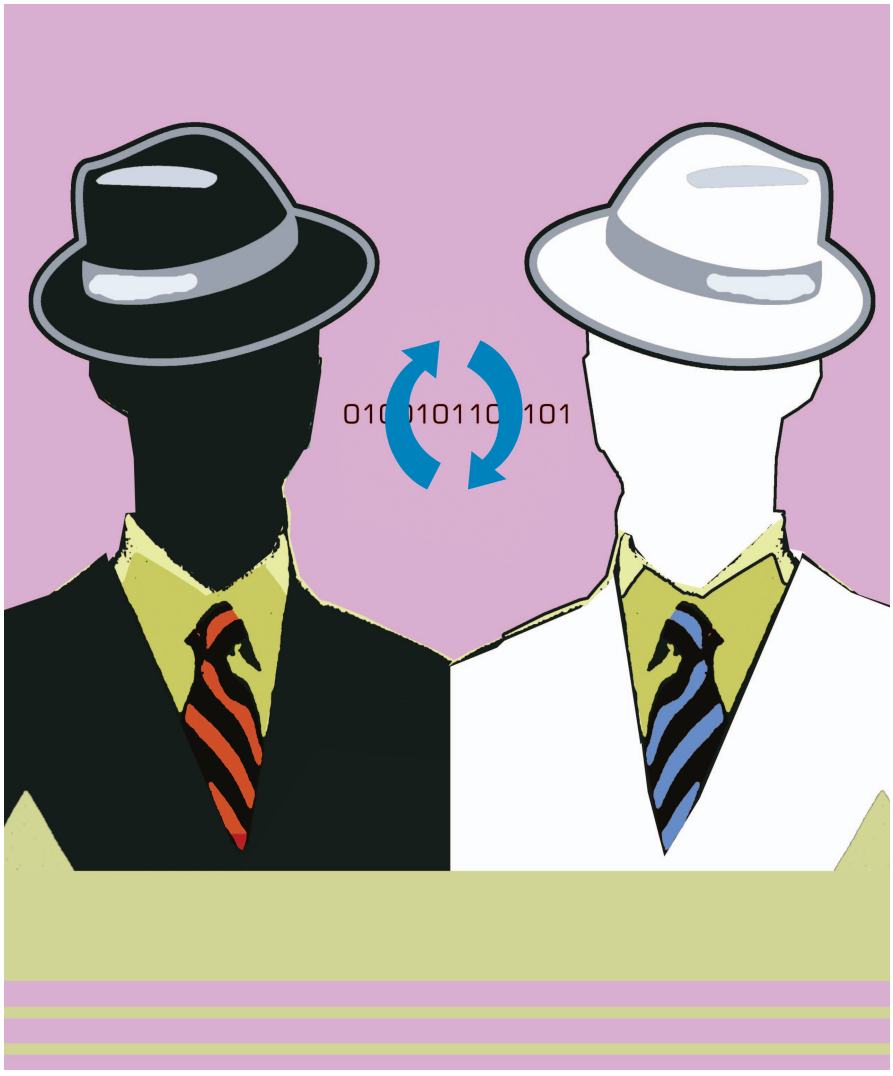
Mohammed Fadel Mokbel

An embellished macro descriptive language for reverse assembly code

Computer programming language, whether general or domain specific, is the perfect path toward a universal standard of mutual understanding among programmers worldwide. In general, people tend to be acquainted with things that are factual and emblematic rather than with those that are nonfigurative or nonrepresentational. The pursuit has been undertaken for an ultimate heterogeneous natural language (NL) that combines executable code syntax with self-describing information.

The revolution of reverse code engineering has invaded most of the software security domains from protection annihilation and malware analysis to legacy systems restoration. This has led to the development of highly advanced and intelligent tool such as the interactive disassembler (IDA), which has become a standard tool among reversers. Even though this was a quantum leap at code reconstruction, there is a need for more elaborative methodologies. A lot of work and attention is required to facilitate the representation of snippet code either graphically or grammatically.

The key feature of this new proposed macro descriptive language (MDL), or substitution language, is based on the preprocessor, macroexpander, macro definition that is used extensively with C legacy code, which is a very simple macro processor. In C++, other possibilities are included and are not restricted to *const, inline, template,* and *namespace* mechanisms as alternatives to many traditional uses of preprocessor constructs. Adopting the official C definition will fulfill most of the textual search-and-replace at the token level, and with C++ mechanisms it will add an additional balance for code representation. It is a domain specific language, designed for a specific set of especially crafted tasks. Moreover, it comprises an extendable set of predefined keywords, using connotative names for variables and special words. These keywords are classified under special categories related but not limited to the main division root in the region of reverse code engineering (RCE). Stating what keywords, terminologies, taxonomies, and nonfunctional words to incorporate in the main code and the rules that



© BRAND X PICTURES & PHOTODISC

Digital Object Identifier 10.1109/MPOT.2009.935245

0278-6648/10/$26.00 © 2010 IEEE

describe how these relational objects should interact with each other is too demanding for a single individual. Setting or proposing a standard in any scientific field requires a master community.

This research is not meant to be a holistic solution for an obscure language [high level language (HLL): e.g., C++] used today in the field of reverse code engineering and assembly snippet code [low level language (LLL): e.g., assembly] nor is it intended to be a replacement for any traditional system. It is a systematic attempt toward an integral standardization, which is to be set primarily by the black hat community members, white hat researchers, as well as different areas of computer specialty. Pragmatically, working on a subset of this colossal field of RCE in computer science would achieve satisfactory results that can be taken into consideration.

The subjects that need to be examined in the future include: nonfunctional keyword insertion, control structures, functions, arrays, pointers, namespace and the binary scope resolution operator (::), each of which is clarified by an example. Some of the C++ private keywords, assignment, logical, equality operators, and others will be replaced by more self-documenting ones, and finally a complete multifaceted case study that will most likely be found in the existent practical scenarios.

The mechanism behind MDL

The driving force behind MDL is from the underlying complexity of examining the assembly instructions. This is an imperative requirement in the software security field and especially in the malware analysis domain, where only very little is known about these malware malicious behaviors in advance. Reverse engineering these malware samples statically requires a thorough understanding of whatever functions are under analysis. In addition, documenting these functions statically is not an easy process, for the reason that these assembly mnemonics (opcode) are very short, usually from one to five letters. The semantics of each assembly instruction alone within specific function boundaries is ambiguous and does not reveal the intended behavior unless it is dynamically examined through the use of a debugging tool. Therefore, the sequence of these instructions in a given function must be structured to unfold the anticipated behavior.

MDL infrastructure profoundly relies on the preprocessor directive macro definitions, which are lines included in the code of our programs that are not program statements but directives for the preprocessor.

This is where MDL plays a major role in defining this transitional phase as shown in Fig. 1. It starts with a compiled executable file in which the source code is not available. After that comes the disassembly phase, in order to probe the algorithm that exhibits whatever behavior, by going through a set of assembly mnemonics. To give a clear idea about the algorithm under assessment, you need to map the algorithm to an HLL such as C or C++ and then translate it to MDL statements (strong translation). It is a very weak translation to go directly from LLL to MDL, since skipping the intermediate HLL translation will lead to a dead list of MDL statements (only comments). Hence, the code will not be executable after all, due to the fact that there is no mapping phase established between the base language (assembly in this case) and MDL.

As previously stated, MDL infrastructure profoundly relies on the preprocessor directive macro definitions, which are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a pound sign (#). The preprocessor is executed before the actual compilation of code begins; therefore, preprocessors digest all these directives before any code is generated by the statements. To define a simple macro, add a new keyword to the syntax of C++ (nonfunctional NF, ignored by the compiler) or to take arguments, the layout should look like Fig. 2.

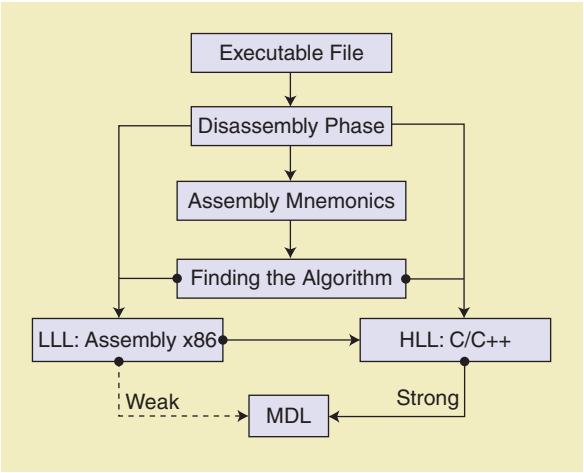


Fig. 1 MDL mapping phases.

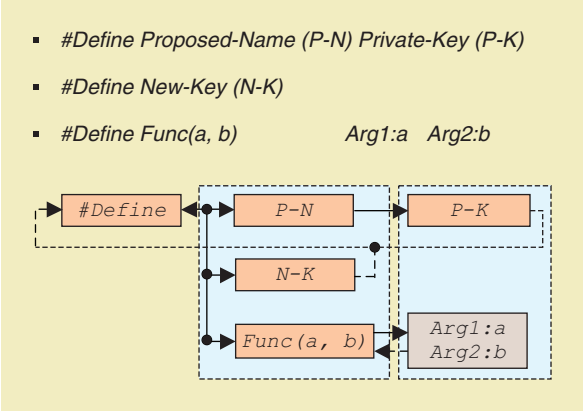


Fig. 2 Define preprocessor directive structure.

For example, substituting the equality operator (+) for a more communicative operator (Plus) would be defined as *#define Plus 1*. When this line appears in a file, all the subsequent occurrences (except those inside a string) of (Plus) in that file will be replaced (expanded) by (+) before the program is compiled and the same goes for adding a new keyword or defining a new function.

This is a brief introduction about #define, and it does clearly and completely carry out all the required operations. However, simplicity comes with a price, since macros know nothing about C++ types or scope rules and only a little about C++ syntax, and somehow it's not easy to manage with error messages or code debugging. Still, most of the operations are safe by design, and they will not interact inadequately or yield unmanageable situations because it is too risky to let it happen.

There must be a consistent set of general principles that control the consistency, stability, and uniformity of MDL when defining the outer layout shell for whatever subject is under plan.

Creating a subtle unification with RCE will set the foundation for all subsequent materials presented in this article. In the everyday scenario, the same process repeats itself, disassembling a binary file, stepping through snippet assembly code, locating the responsible snippet of whatever behavior under investigation, and possibly going under heavy translation from LLL to HLL. These highly advanced procedures are not easy to master or put into practice as it may seem. The continuous cycle of these disciplined approaches are shown in Fig. 3.

Figure 3 shows the status of each language in terms of description and complexity level. As you go upward (the lower half), the level of description (standardization, well-documented description of an application's internal data) in each language is increased by a factor relative to its area; it's obvious how this area for each language is getting wider as you move upwardly, where high descriptive language (HDL) is the most informative. These left-right arrows indicate a strong relationship between HLL and HDL because they are inter-mixed in almost every step. On the other hand, as you move downward (the upper half), the level of complexity increases oppositely with respect to each language description level beneath it directly (asymmetrical relation). Working forward at level 1 ↔ 3 to attain a reasonable proposal would decipher most of the cryptic terms used today in the RCE research community.

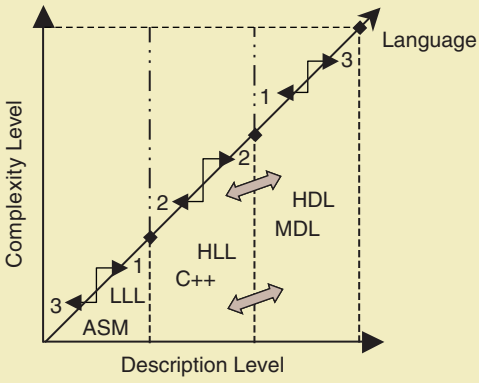


Fig. 3 LLL, HLL, and MDL asymmetrical relation.

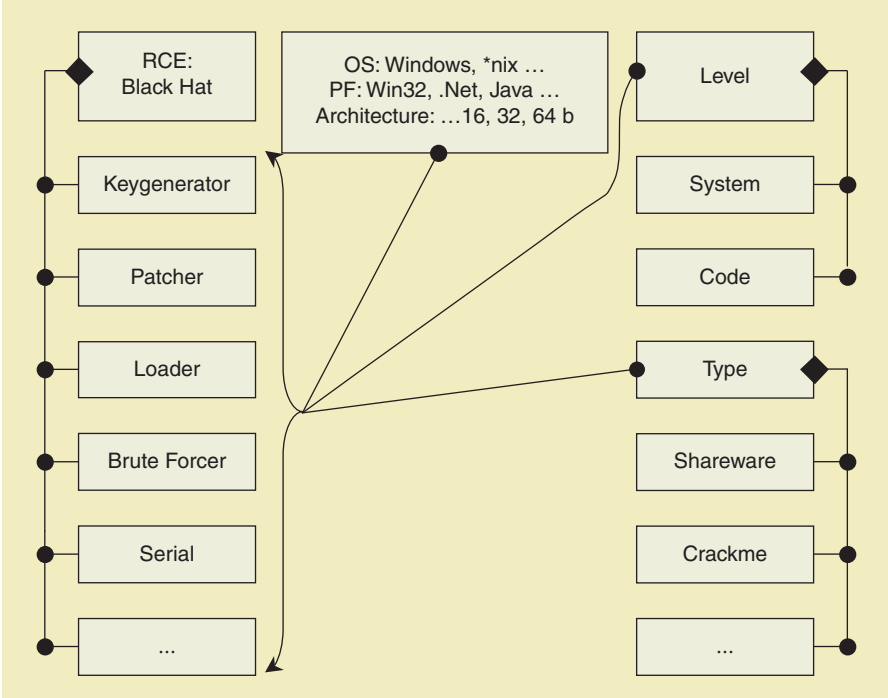


Fig. 4 A miniature hierarchy of RCE: Black hat subject.

The integration between C++ and pseudocode writing is very simple to learn and easy to use, and in no way does it interfere with one's learning of an actual programming language. The reasoning behind this mechanism is that the integration should be scientifically and logically anatomized, even though it does violate what has already been defined—that a pseudocode algorithm is not a computer program. This merging process is not chaotic or lamentable to implement since it follows flexible regulations that perfectly adhere with HLL.

Subjects that need to be examined

RCE is a wide-ranging spectrum field of study to be entirely stretched out. What are these subjects? What measures and procedures should be taken into consideration when setting the rules that govern the overall structure of MDL?

In general, the topics are extremely synthesized with each other depending on the case under analysis, in other cases they tend to be more contained and less appendaged and self-coherent. Some of these topics are: encryption/decryption, obfuscation/deobfuscation, reversing/antireversing, crypt analysis, and malware analysis. Each one of these topics is subdivided into more detailed related processes and techniques. Writing a complete structured analysis for each one of these topics is beyond the scope of this article. Discussing a specific area of RCE would shed some light on how things should be done. The black hat communities are more engaged in this revolutionary world of RCE. Figure 4 demonstrates how they are grouped and divided. This is a very compact overview of the RCE Black hat subject, because every subject matter is subdivided into multibranches (not shown).

There must be a consistent set of general principles at one side that control the consistency, stability, and uniformity of MDL when defining the outer layout shell for whatever subject is under plan. On the other side, there must be more strict principles and rules that manage the relational flow through MDL statements execution with an ultimate security that prevents the inconsistency

between base language and MDL keywords, and at the same level preserving the consistency between MDL libraries. Agreeing on a relative supportive strategy that is expandable in the future with an additional matching schema would clear up the hazy layer that covers the environment of this language. Below are some of the guidelines and characteristics classified with respect to RCE, specifically the assembly snippet code that should be taken into consideration when programming using MDL:

- MDL declarations, syntaxes, keywords, and taxonomies should be clearly identified and self-definite to bring self-similarity, self-consistency and existence. They should be well known to the subject matter and follow a recognizable and predictable pattern, particularly when injected in the middle of a complete statement.
- A synonymic and polygamy set of keywords should be used interchangeably whenever needed to complete the intended meaning.
- The integrity of MDL statements should be preserved in an associative manner without interference with base language (C++) since this could lead to abnormal behavior.
- Characterized libraries for each subject should be separated and given a self-documentary name (e.g. LogicalOprtr.h, ArtithmaticOprtr.h, RelationalOprtr.h, SupportiveFunct.h, AssignmentOprtr.h, EqualityOprtr.h, EscapeSeq.h) with a master library that includes all the slaves named, something similar to MDL.h.
- Proposed taxonomies and keywords should be mutually exclusive with respect to library identification (they must not go beyond their own library), comprehensive, unambiguous (understandable and defined to avoid any confusion during classification), acknowledged (logical and intuitive so that they could become generally approved), and constructive (could be used to gain insight into the field of inquiry).
- Closeness of mapping: MDL environment should have a high level of expressiveness to relate the operations in the problem to corresponding operations in the program domain.

There are other factors that should be taken into account, varying from low alpha first order principles, which does not involve complex sequential relations, to a higher order of adaptive code, which does involve complex relational inheritance chains that evolve in response to multi-parameterized definitions, all of which are put together to mimic complete code outline.

The emphasis is on code snippet with a transformation phase from a low level to high level language that needs to address a complete, comprehensible, and fully functional procedure.

This is not to be integrated in large projects because it is time consuming and inefficient. It is still partially applicable where considered necessary because a plethora of extra declarative statements needs to be inserted with many other aesthetic correlations. Mainly, the emphasis is on code snippet with a transformation phase from a low level to high level language that needs to address a complete, comprehensible, and fully functional procedure. As a result, the outcome of the final transformation is a newborn NL code that resembles one with high level thoughts. This scheme follows a reverse mode of what already has been discussed in a natural language processing for natural language programming paper, which proposed a system that attempts to convert natural language text into computer programs by Rada Mihalcea, Hugo Liu, and Henry Lieberman.

Proposed schemes for MDL code representation

Writing a complex indicative code by means of core code block emulation from LLL to HLL imposes establishing an advanced, well-equipped workspace armed with a lot of graphical code representation tools and evolvable C++ classes that are to be incorporated in the main code framework. In this article, two schemes are proposed in philosophical terms for this task in order to smooth the progress of writing and visualizing code symbolization in correspondence with the output readings.

Verbose analytical transparency scheme (VATS)

The key feature of this proposed method is the relational input output reading system. Instead of writing a disconnected, invisible generator to the printed statements (especially in console mode, e.g., Keygenerator, a serial number will be generated to the entered name without noticing any procedure of code calculations, which has been done inside the box), a better method would be to uncover all these reckonings behind the curtain and put everything from the basic statement to function analysis outside the box in a logical order, followed recursively by LL and HL code itself. Every code statement should be clearly identified and marked as a possible functional task that draws a parallel and more enhanced descriptive version of the same instruction being involved. This could be described as a live debugging scenario, but in this case everything is managed and designed to be fully expandable, where capturing and documenting each loop variables are done in a controlled environment by sending the outputs either to a preprogrammed HTML template file or directly to the screen (console or graphical user interface). Apart from these suggested methods, there must be a translucent communication system that consists of different groups of classes or functions that serve as a silent interactive structure where every executable statement is logged dynamically and inherently.

Using either deterministic or probabilistic methodologies that belong to mathematical formulations will just reconcile the fitted outcomes, all of which are clustered in response to its functionality. Hence, the analogy is a base skeleton that carries the entire compulsory tasks efficiently and automatically while preserving the reliability of code ramification via a precoded set of modules or classes using object oriented programming (OOP) concepts. It is easier to write a snippet code using this scheme if MDL is used in first place because the level of code comprehensibility is much more elegant than mere simple abstract statements.

As a proof-of-concept (POC), I have coded a small utility in console mode (VATS v0.1) to adduce my theory in which a full-scale detailed analysis of serial checking algorithm (SCA) is being addressed and mapped to an HTML template file like live debugging analysis. This POC utility is programmed using the C++ language (Microsoft Visual Studio 2005). The sample (SCA) used is

> OOP should be used as a main framework with an intelligent engine and parser that could decide what pre-coded template to use based on SCA evolution and pattern with the aid of special marks.

from "KeyGen-me N°1 by devilz's." Unfortunately, the current POC is not intelligent enough to accept other scenarios; its modularity is limited to the embedded SCA. To make it modular and less restrictive to adopt a different scenario, OOP should be used as a main framework with an intelligent engine and parser that could decide what pre-coded template to use based on SCA evolution and pattern with the aid of special marks used as indicators for variables, functions, and control structures. There are other options that could be added to increase the level of control, recognition, and visual enhancement using XML or HTML to improve the overall code linkage.

Figure 5 shows the disassembled instructions of the main SCA plus a brief comment to the right of each. You can see that these sets of instructions are not expressive in any way nor do they tell anything about the functional behavior of SCA. Because of that, VATS tool (available at http://www.themutable.com) is designed to profile the dynamic behavior of SCA. The output is a full-fledged live debugging analysis of SCA, as shown in Fig. 6.

Figure 6 shows the HTML template used to document the runtime functional behavior of SCA (Fig. 5). This is automatically generated based on the pre-coded HTML template. The HTML template is generally limited to what SCA can do. On that account, VATS profiles SCA execution in an organized and controlled test bed. The coverage analysis of the source code is almost similar to the structural testing technique in which you test program behavior against the apparent intention of the source code.

In addition, a theoretical MDL version of the SCA is also presented along with C++ for completeness to show the elegance of MDL representation. Using Nassi-Schneiderman diagrams—main (Fig. 7):

A Fully Descriptive Analyses of **S**erial **C**hecking **A**lgorithm (**SCA**)				
Hotspots	004010E2	6A 0C	PUSH 0C	Count = C (12.) (*)
	004010EE	Call GetDlgItemTextA(008C04BC, 64, 00403380, 004010E2)		
		+ Check if the size (EN) in EAX is # from zero; if not GOTO NL		
	00401105	Call MessageBoxA(NULL,Fill in the blank, The name please !!!, Null)		
	0040111F	Call GetDlgItemTextA(008C04BC, C8, 00403380, 00401136)		
		+ Check if the size (ES) in EAX is # from zero; if not GOTO NL		
	00401136	Call MessageBoxA(NULL,Fill in the blank, The serial please !!!, Null)		
	NL: Next Line, EN: Entered Name, ES: Entered Serial, (*): ValidLength(EN) = 12-1 ('\0')			
	+ BEGIN (MotherShip of SCA)			
	00401150	33D2	XOR EDX,EDX	EDX = 0;
	00401152	33DB	XOR EBX,EBX	EBX = 0;
	00401154	33C9	XOR ECX,ECX	ECX = 0;
	00401156	33C0	XOR EAX,EAX	EAX = 0;
	00401158	BE 80334000	MOV SI,KeyGen-m.00403380E	ESI = EN;
	0040115D	8A1C31	MOV BL,BYTE PTR DS:[ECX+ESI]	BL = First Letter of EN;
	00401160	03C3	ADD EAX,EBX	EAX = EAX + EBX(BL);
	00401162	41	INC ECX	ECX++ (Counter);
	00401163	80FB 00	CMP BL,0	Is(BL == 0) (End of Array?);
	00401166	75 F5	JNZ SHORT 0040115D	If (Not): GOTO 0040115D
	00401168	BA 28000000	MOV EDX,28	Otherwise EDX = 0x28;
	0040116D	F7E2	MUL EDX	EAX = EAX * EDX;
	0040116F	83C0 19	ADD EAX,19	EAX = EAX + 0x19;
	END (MotherShip of SCA)			

Fig. 5 Main SCA assembly instructions.

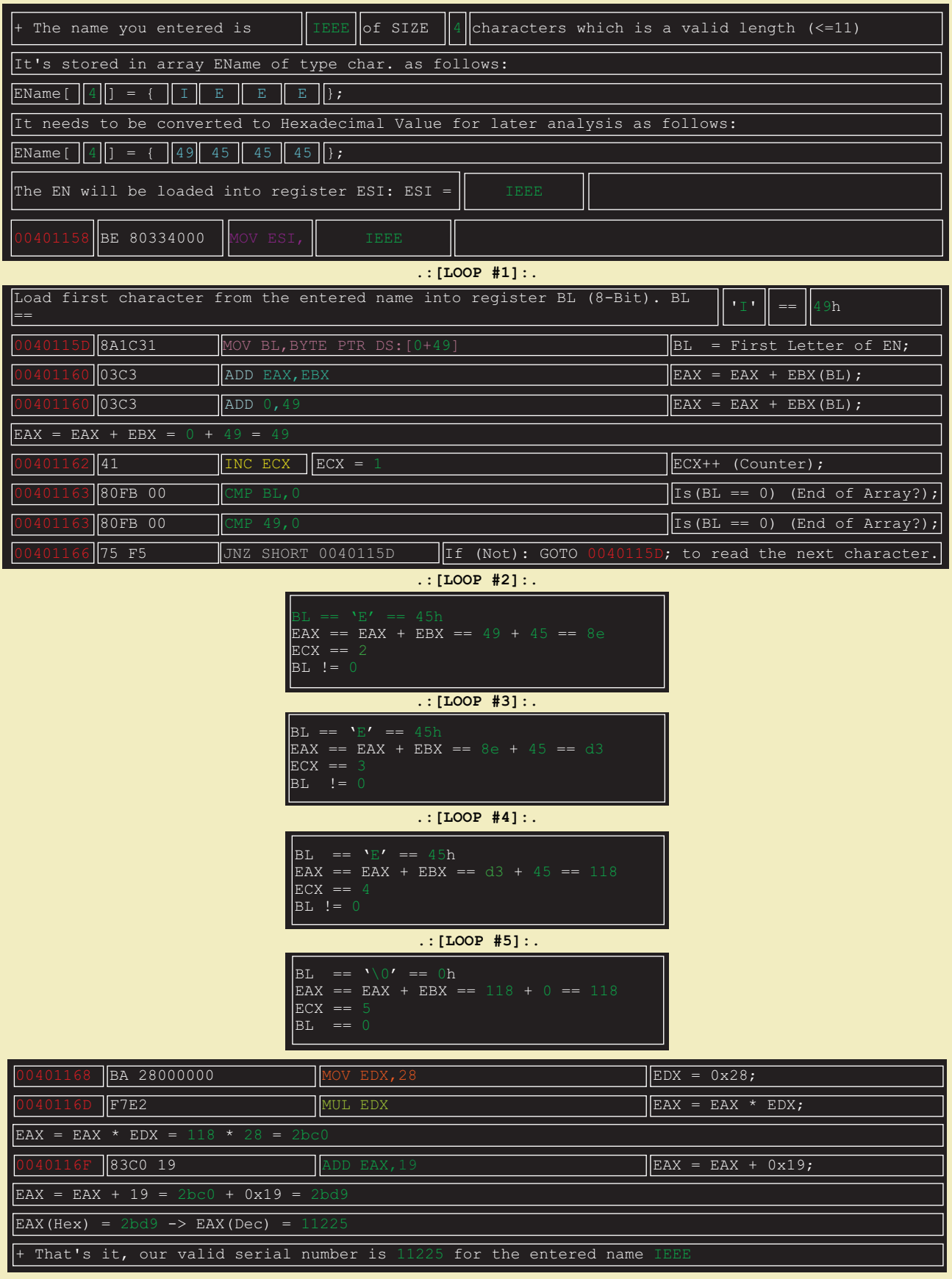


Fig. 6 VATS mapped analysis (storyboard).

```
float EAX = 0;
float EBX = 0;
int   ECX = 0;
float EDX = 0;
const int Size = 12;
char Name[Size];
cout<< "Please Enter Your Name (Maximum 11):";
cin.getline (Name,Size,'\0';

For: ECX = 0 ; Name[ECX] != '\0' ; ECX++
                EBX = Name[ECX];
                EAX = EAX + EBX;

EDX = 0x28;
EAX = EAX * EDX;
EAX = EAX + 0x19;
cout <<dec<<"Your Serial Number Is: "<<EAX;
return 0;
```

Fig. 7 Nassi-Schneiderman diagrams – main(SCA).

```
#include <iostream>
using namespace std;
int main()
{
   float EAX = 0; // XOR EAX,EAX
   float EBX = 0; // XOR EBX,EBX
   int  ECX = 0; // XOR
ECX,ECX
   float EDX = 0; // XOR EDX,EDX
const int Size = 12; // PUSH
0C ; Count = C (12.)
char Name[Size];
cout<< "Please Enter Your
Name (Maximum 11): ";
// MOV ESI,KeyGen-m.00403380
cin.getline(Name,Size,'\0');
   for (ECX=0; Name[ECX] !=
   '\0'; ECX++)
   {
   // MOV BL,BYTE PTR DS:[ECX+ESI]
      EBX = Name[ECX];
      EAX = EAX + EBX; // ADD
      EAX,EBX
   }

   EDX = 0x28;    // MOV EDX,28
   EAX = EAX * EDX;  // MUL EDX
   EAX = EAX + 0x19; // ADD
EAX,19
   // wsprintfA(byte_40339A,
      "%d", (eax * 0x28) +
      0x19);
      cout <<dec<<"Your Serial
       Number Is: "<<EAX;
      return 0
}
```

A good understanding of the relations among the code statements could lead to a better conceptualization.

This C++ block of code could be paraphrased into a more natural language:

Load It
Mother-Ship
- ❖ BEGIN (Turn ON Engine)
- ✓ .Step |I|
- ○ Let Variable EAX, EBX of Type float Equal Zero.
- ○ Let Variable EDX of Type float equal 28 Hex
- ○ Let Variable ECX of Type int Equal Zero.
- ✓ .Step |II|
- ■ Create an Array Name of Type Char of Size 12
- ■ Send Message "Please Enter Your Name (Max 11 :)" to the Screen
- ■ Read Name Then Press Enter
- ✓ .Step |III|
- ➢ As Long As Name Is Different From the Null Character Keep Adding Each Element of Name to Variable EAX
- ✓ .Step |VI|
- • Multiply EDX by EAX Then Save the Result Into EAX
- • ADD 19 Hex to EAX Then Save the Result Into EAX
- • Convert EAX to Decimal
- • .Step |V|
- ■ Send Message "Your Serial Number Is:" Containing EAX to the Screen
- ❖ END (Turn OFF Engine)

This version is easy to convert to MDL because it only requires locating the functional and nonfunctional keywords to define the substitution and rearrange the code structure following the aforementioned guidelines. This will enhance VATS dynamicity, readability and understandability to break the stationary flow of SCA into a more eloquent stream.

Drag and drop scheme

Imagine a complete fully functional code, simple or compound, distributable as a standalone Windows application being designed without writing any line of code. This is one of the best schemes to be adopted as a major evolution in code representation because of its versatility and adaptability. This is not an unrealistic approach; on the contrary, high-quality software named A-Flow is already developed.

Instead of writing all the code you used to write again and again, a better approach would be to choose a different path by using a powerful general-purpose software development and authoring tool because most of the functions are already coded as a building block and ready to be inserted into a new module. Code "building block" visualization is more apparent and the relational flow between the building blocks is more controllable because of the drag and drop feature and the lines used to connect them.

It would be a great achievement if a special tool like this is designed for an RCE integrated development environment (IDE) with a software development kit to add a new functionality as a plug-in or has its own script language (e.g., MDL) to define a new building block built-in function (fully customizable) categorized in a way that will accept different scenarios of RCE field. As shown in Fig. 8, each box holds a built in function designed for its own purpose and they communicate with each other through connection lines. This is neither a perfect representation nor a complete demonstration, it is only a prototype.

Conclusion and future work

This article illustrates both theoretically and practically how LLL, HLL and MDL could be fused together to shape the elementary code structure into more approachable, elegant, and sophisticated delineation metastatements. Further work must be done in the area of code symbolization and interrelation to achieve an agreeable scheme.

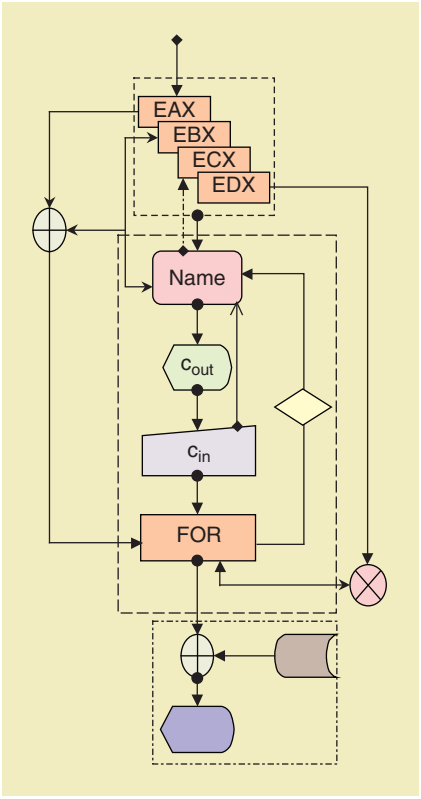


Fig. 8 Drag and drop scheme representation.

The development and improvement on MDL will continue from many different aspects regarding code keywords classifications by proposing appropriate substitution keywords, according to the principles mentioned above. A free tool called Notepad++ will be used as an IDE for MDL syntax highlighting keywords, syntax folding keywords, comment keywords, operators, and implementing a customized auto-completion feature for MDL keywords. For up-to-date information about MDL, visit http://www.themutable.com. This is an example of how MDL could be formulated in the future:

```
  If (EAX == EBX ? EAX : EBX);
To
  .Step|1|
IF LP EAX Equal EBX Therefore
EAX Otherwise EBX RP;
```

Acknowledgment

The author would like to thank all the RCE communities for their contributions.

Read more about it

- IDA pro advanced: Interactive disassembler, V5.0.0.879. Datarescue [Online]. Available: http://www.datarescue.com.
- M. Rada, L. Hugo, and L. Henry, NLP (Natural Language Processing) for NLP (Natural Language Programming) *Proc. 7th Int. Conf. CICLing 2006*, A. Gelbukh, Ed. 2006, pp. 319–330.
- S. Juan. (2006, May 16). *Preprocessor directives* [Online]. Available: http://www.cplusplus.com
- S. Bjarne, *The C++ Programming Language*, 3rd ed. Reading, MA: Addison Wesley Professional, June 1997.
- K. Bernard, B. Robert, and R. Sharon, *Discrete Mathematical Structures*, 4th ed. Englewood Cliffs, NJ: Prentice Hall, 2000.
- S. W. Robert, *Concepts of Programming Languages*, 7th ed. Reading, MA: Addison-Wesley, Apr. 2005.
- H. John and L. Thomas, "A common language for computer security incidents," Sandia Nati. Lab., Sandia Rep. SAND98-8667, Oct. 1998.
- C. Steve. (1996). Code coverage analysis. Bullseye Testing Technology, {Online]. Available: http://www.bullseye.com/coverage.html.
- G. Rick, "Code profilers choosing a tool for analyzing performance," Freescale Semiconductor, Doc. No. CODEPROFILERWP, Rev. 0 11/2005.
- M. F. Mohammed. (2006). Reverse code engineering: Emphasizing on breaking software protection. B.S. dissertation, Dept. Comp. Eng., Lebanese Int. Univ., Saida, Majdelyoun [Online]. Available: http://www.themutable.com. An updated version (exclusive edition) as of June 29, 2006, under the pseudo name tHE mUTABLE.
- P. Alexey. (2003). A-flow software applications visual designer, V3.50.00 [Online]. Available: http://www.aflow-designer.com
- H. Don. (2007). Notepad11, V4.1.1 [Online]. Available: http://notepad-plus.sourceforge.net

About the author

Mohammed Fadel Mokbel (mfmokbel @ieee.org) received his B.Sc. degree in computer engineering from Lebanese International University, Lebanon, in 2006. He is pursuing his master's degree in computer science (HPGC Research Lab) at the University of Windsor, Canada. His research interests include reverse code engineering, software security, Web engineering, combinatorial optimization, grid computing, and parallel computing. He is a Graduate Student Member of the IEEE.