Dynamic Syntax Tree: Implementation Results

Prof. Tim Moses, Department of Software Engineering, BitBrainery University, London – UK
David Syman, CTO, Security Review Консультант, Chişinau - MD
Marco Barzanti, Security Auditor, Poste Italiane - IT
[2012- 10th of December]

Abstract **–In our earlier research[1], we described the *Dynamic Syntax Tree* method implementation for enhancing the Static Analysis process. After 10+ years of experience, we collected the significant results presented in this paper**

Keywords **- dynamic syntax tree, dynamic analysis , static code analysis, abstract syntax tree, parser, semantic**

I. INTRODUCTION

In our earlier research[2], we presented a ***Dynamic Syntax Tree***-based implementation. Main differences respect than ***Abstract Syntax Tree*** (AST) are:

- *Very compact Syntax Tree*. More than 10 time less than AST, due to adding an *Object Dictionary* containing all object information belonging to a Class, to the DST itself that contains only pointers to the Object Dictionary.
- *The Syntax Tree is split into a number of small DSTs*, and paged to small XML files, reducing RAM consumption. In this way a fixed peak RAM value can be configured before Static Analysis execution.
- *The Syntax Tree* includes Dynamic information too, for more accurate Analysis results.

The main contribution of this paper is to present 10 years results collected in real, international-wide business cases. For testing the implementation, we use the re-engineered Security Review Консультант products, named Security Add-on and Quality Add-on for McCabe®, implemented with the Dynamic Syntax Tree. Dynamic code is processed by mapping of dynamic constructs, and then usual techniques for vulnerabilities detection in the static way are used in combination with dynamic sandboxing. The semantic analysis works even for static languages too. The implementation is able to gather useful information about the source code, such as possible values of variables or possible relations between objects.

II. DYNAMIC SYNTAX TREE IMPLEMENTATION

Pre-processing the source code created a specialized Dynamic Syntax Tree for each Class found. That has been applied to traditional programming languages too, like COBOL, where Classes are represented by Programs, Methods by calls/Performs, Parameter by Using etc. In the Security Review Консультант implementation, this pre-processing phase will generate:

- *A separate* ***Object Dictionary*** for each Class. All Class objects will be mapped into 2-bytes Dict-Id, handling a maximum of 65535 objects per Dictionary. Instead of storing the object name, 4-bytes and 1-byte pointers to source will be used for retrieving the object's name (source code line and name's starting column). Parent Dict-ID (for child Classes) or 1-byte Type + 1-byte local/global attribute (for the others), and 1-byte bitmask Attribute field (abstract, serializable, public, private, protected, static and final).
- A **Dynamic Syntax Tree** for each NameSpace or Package storing: 10-bytes NameSpace or Package Name and File Name (including web pages and configuration files) in compressed format. Further, 2-bytes Dict-Id for Class, Inner Classes, External Classes, Methods, Parameters, Branches and Variables will be stored in the tree. For Methods and Branches, further to Dict-Id, also an hash code will stored, for Code Duplication detection purposes. For Branches, conditional statement as a single line and nesting level (for calculating Quality metrics) are also stored. Fields will be compressed using *Huffman Coding* [3].

Thereafter this pre-processing enables us to work with the syntax tree of the dynamic source code as it is in a static code with some limitations, that are not resolvable until runtime in dynamic languages. For that reason we provide a binaries analysis too. Binaries will be *sandboxed* collecting dynamic information at runtime, using a very fast algorithm that we discussed in [4]. Mixing source code and binaries analysis fixed the above mentioned limitations, updating the Dynamic Syntax Tree with additional information. Object Dictionaries and Dynamic Syntax Trees are multiple, and optimized for low resource consumption and higher performances.

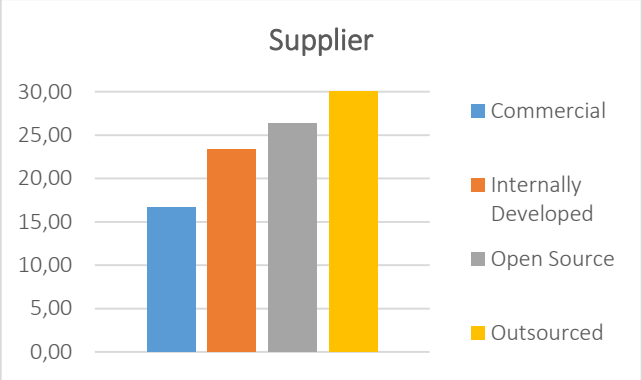
Differently from AST and CST (Parse) trees, in case of huge Classes having more than 65535 objects, the DST Object Dictionary structure (68,083 nodes), is paged into some small XML files, about 575KB sized each. The same is done with the Dynamic Syntax Tree itself: only 4096 Classes at time are processed, max 135KB each. There was no case of RAM usage over 700MB, that means we were able to perform a Static Analysis using a low-end Windows XP notebook with 1GB of RAM with a single Pentium processor, and up to 5 simultaneously static analysis of different applications at time are achieved using only a 4GB RAM, in a dual-core processor machine. A Windows 2008 server version with 8GB RAM, processed up to 15 analysis at time.

III. RESULTS

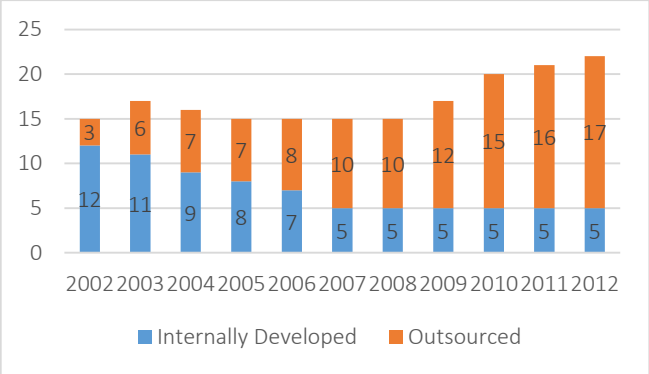
The re-engineered software was used for analyzing about 800 Millions of Lines Of Code (MLOC), in different business sectors, located in 5 countries:

Results were collected in anonymous way, only some technical information were stored like Industry sector, Supplier (outsourcer) type, target Platforms and Languages.

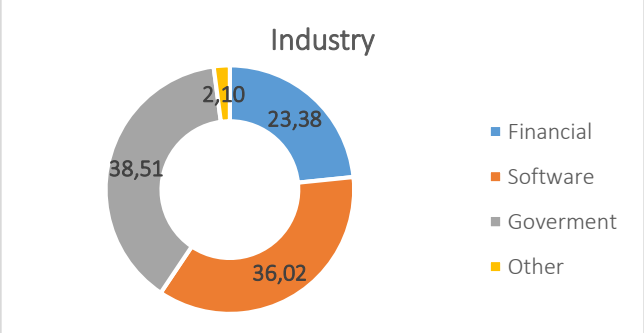
Each Application has been analyzed in 3-4 Versions per year, over 10000 analysis in total.



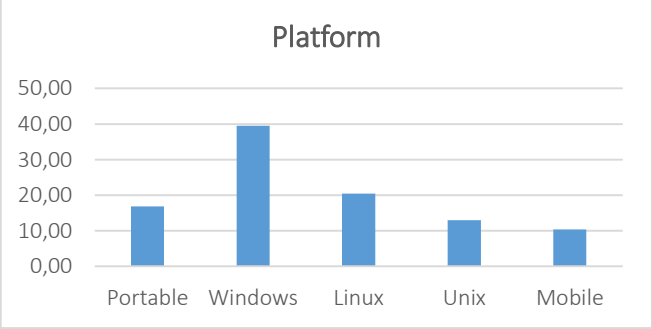
Each Customer has more than 2 Suppliers, 30 in total, some of them in common.
Many Suppliers are products' owners (Commercial Supplier).



% of Internally Developed projects are growing down versus Outsourced.

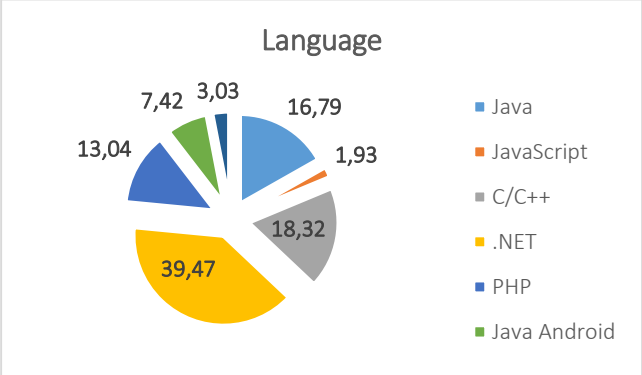


Government Institutions and Software integrators were the main Customer's base of our analysis, even Financial Institutes (Banks, Assurance) were the most contributors in terms of application size (MLOC, Millions of Lines Of Code).



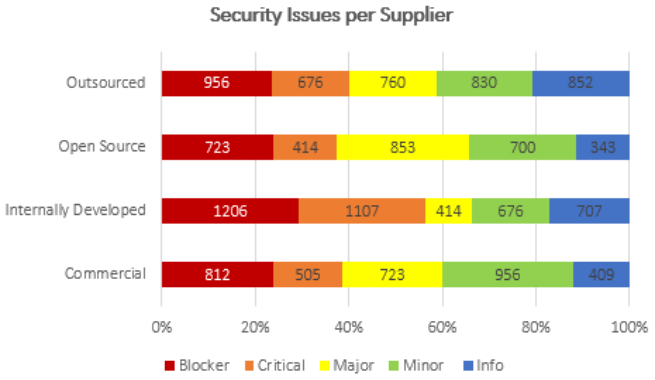
Portable applications are mostly written in JAVA, JavaScript and PHP, but only few are truly portable as reported.

Mobile Platforms are emerging.

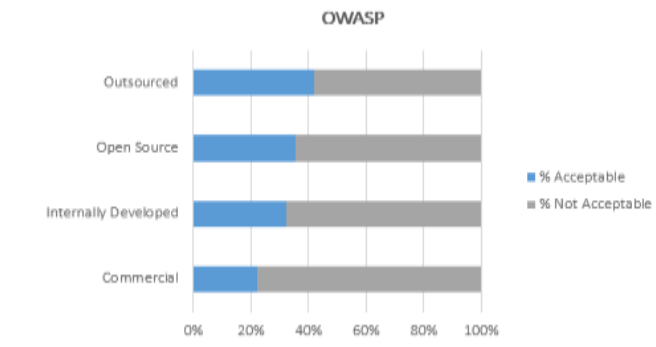


High Business Criticality does not drive all development projects "in-house". More than 30% of all applications rated High or Very High in business criticality were sourced by Commercial software vendors.

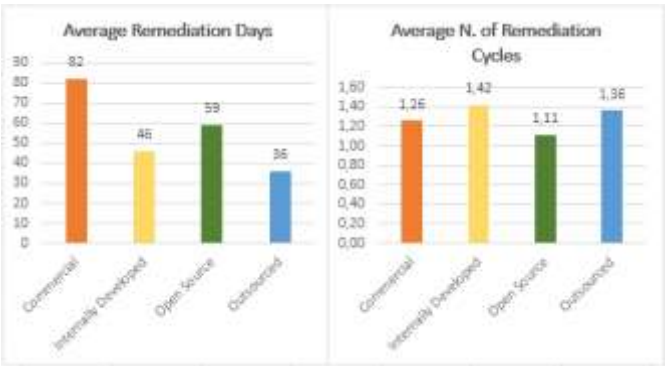
Majority of projects are not compliant with OWASP Top 10 or SANS Top 25. Open Source has higher quality than Internal



Developed.



Commercial has longest remediation cycles



The following table synthetizes our effort, showing the huge difference between the number of Dynamic Syntax Tree (DST) nodes obtained with the described implementation, respect than number of Abstract Syntax Tree (AST) nodes calculated with ANTLR[5]. We are working on further 15% DST nodes optimization, 20% expected.

Syntax Tree	#	%	AST	DST	MLOC
Java	505	16,79	1069044204	111358771	236
JavaScript	58	1,93	82443083	5748024	16
C/C++	551	18,32	672436662	70208004	149
.NET	1187	39,47	769361183	83424707	221
PHP	392	13,04	390707047	49948344	106
Java Android	223	7,42	272779120	59986149	60
Objective-C	91	3,03	111055783	11595151	25
	3007	100,00	3387827081	392269150	812

Is the number of analyzed applications. 3007 in total
MLOC is Millions Lines Of Code. 812 MLOC in total.

IV CONCLUSION AND FUTURE WORK

The presented paper described 10 years of analysis results obtained us an implementation of automatic analysis of the dynamic language source code using *Dynamic Syntax Trees*.

The implemented Dynamic Syntax Tree will be used for some product's re-engineering and, after some years of stable Static Analysis experiences, will be compared to other AST and CST-based solutions. A separate paper at that time will be available.

V. ACKNOWLEDGMENTS

This work was gently supported by:
Ruth Goldberg, Software Engineer, Security Review Консультант, Chişinau - MD

REFERENCES

[1] Moses.T., Syman.D., Barzanti M. Static Analysis: A Dynamic Syntax Tree Implementation. London, December 2001

[2] Moses.T., Syman.D. Static Analysis of Applications written in modern languages. Moldova, 1999. Translated from Russian and published by ResearchGate, 2008

[3] Huffman D.A., "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., September 1952

[4] Moses.T., Syman.D., Barzanti M. Binary Analysis: A Dynamic Sandboxing Implementation. London, July 2006

[5] T. J. PARR University of Minnesota, R. W. QUONG School of Electrical Engineering, Purdue University. ANTLR: A Predicated LL(k) Parser Generator. July 1995.