Review Paper: Effective Code Change Quantification for Defect Estimation

Usman Ahmed Shami (usmans@u.washington.edu), Junaid Ahmed (jun.ahmed@gmail.com)

1. Introduction

Program change analysis involves the effective calculation of code units that are fundamentally different in different versions of programs. This can happen between successive versions of binaries or between two milestones in a release phase. The kind of changes also needs to be determined so that we can filter out useless noise in our analysis.

There are multiple ways to address these issues and each of these has their own advantages and disadvantages. In this report, we aim to analyze a few of these techniques from an algorithmic perspective and to critically examine the consequences of using each of those.

Once we have the code change metrics in place, we can use statistical regression co-efficient to come up with metrics that can be used to predict defects early in the release cycle of a particular product. This will cover one of the major implementations of the quantification of code change.

2. Applications of Code Change Analysis

The successful detection and quantification of code change has got quite a number of useful applications, a few of these have been described below:

2.1 Test Optimization

Test execution cycles can be optimized across successive releases by executing only those test cases that are found to hit the code paths that have been altered.

2.2 Profile Propagation

This is the basic summary of code usage statistics. To characterize the program behavior proper profile data is required. To effectively manage changing code profiles is not trivial as thousands of lines of code might be changing in brief periods. So, effective and fast binary code change analysis can also help in code profile change management. [1]

2.3 Regression testing

Regressions are internal bugs caused by code change before it is shipped. During all the phases of the SDLC (software development life cycle), regressions guide the milestones and software quality. It is easy to see that the determination of software change leads to lower costs in regression finding and fixing. [2]

2.4 Software Version Merging

This deals with refactoring of code and versioning of subsequent code releases. Metrics are also usually needed to determining the amount of “real” code churn taking place between different versions.

2.5 Software Security

Researchers need to determine the binary changes in the binary releases to know if the change can be determined by reverse engineering and can be used to exploit systems that have not been updated

3. Binary Code Change - The problem and challenges

We analyze binary change in comparison of source code changes as it is a more reasonable estimation of logical code changes.

The problem of matching binaries i.e. a set of instructions, set of basic blocks and set of functions in either two versions of the same or similar executables can be defined as:

Suppose that a executable E' is created by modifying E. Determine the difference Δ between E and E'. For a fragment f' ∈ E', determine whether f' ∈ Δ. If not, find the corresponding origin of f' in E.

and / or

Suppose that we have executables P and Q. Find all fragments f' such that f' ∈ P and f' ∈ Q.

The above two problems i.e. comparing different versions of the same executable or two arbitrary executables that contain an amount of common blocks is a difficult problem due to compilation and optimization differences at the instruction level. Some of these challenges are:

- Two binaries might appear different because different registers were allocated.
- Due to compiler's generation of instructions according to pipeline of the processor, the instructions might be in a different order. Thus, if the same code essentially exists in two executables, it might appear out of order in one.
- Due to change in numerical offsets in memory address operands. These offsets often change from build to build due to changes in data layout
- Compiler optimization leads to branch inversions, in lining functions etc.
- Including binary changes that are direct
- Ignoring binary changes that are indirect

4. The Graph Isomorphism Approach

The key point in the algorithm is to model the executables using graph theory and analyze the levels of isomorphism in the graphs of the executables. [3] [4]

4.1 Algorithm

Given: Two executables named *Old* and *New*

4.1.1 Preprocessing:

The executables are preprocessed to obtain a good disassembly for the same. Let the disassembly of Old be denoted by O and the disassembly of New be denoted by N. There are many tools available for this in the industry (for e.g. [5]) and this process is a different class of problems and is not detailed here.

4.1.2 Setting up Graph Representation:

Disassembles mentioned above are represented as a directed graph of graphs. The nodes in the directed graph represent the disassembly of C functions in the original source code. i.e. F = $\{f_1, f_2 \ldots\ldots f_n\}$. There are directed edges from a node u to a node v if the function f_u calls the function f_v.

Each function $f_i \in F$ is further represented as a control flow graph. Control flow graph is a data structure used to represent a procedure in compilers where each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow.

Let us call the graph obtained from O as G_o and that from N as G_N

Let us also define some additional notation.

Power Set: For any given set H, the power set of H, written here as □(H), is the set of all subsets of H. Thus in our case the power sets of G_o and G_N are □(G_o) and □(G_N) respectively

Parent (v): A function Parent (v) which takes as an input the vertex v and returns the parent u of v.

Selectors(x, G_N): Selector (x, G_N) defines for a given vertex $x \in G_o$ of a graph, and set of vertices in another graph G_N, a mapping that returns either one node or nothing (empty set). The one node returned is a node from G_N that is most similar to x. If there are multiple nodes in G_N with certain closeness, the mapping returns an empty set (nothing). Some common selectors can be:

- In-degree of the nodes: i.e. select nodes if their in-degree is the same.
- Recursive Nodes: i.e. select nodes that have a link to themselves
- A 3-tuple (α, β, γ) for a control flow graph where α represents basic blocks in the function, β represents the number of edges linking them to form a CFG and γ represents the number of sub function calls in basic blocks

Property(G_o, G_N): Property(G_o, G_N) is a mapping that maps for two graphs G_o and G_N with their certain subset of nodes G_o' and G_N' . If we have multiple properties available, we call the set of properties $\hbar = \{\hbar_1, \hbar_2 \ldots \hbar_k\}$ This sort of mapping reduces the size of the sets used by selector to better predict a isomorphism and return a non empty set. Some common properties are:

- Nodes having same name in the call graphs. These can available as either debug information or import/export information.
- Nodes having similar/common string references.
- Nodes having same small prime product (see section overcoming challenges)
- Nodes having same sub function calls in control flow graphs

4.1.3 Graph Isomorphism Algorithm:

- Given a Selector, the algorithm constructs an approximate initial graph isomorphism p_1 for the graphs G_o and G_N. The algorithm for this is given as:

```
For ħi ∈ ħ  where i  = 1 ....k do
       Obtain (Go', GN') from ħi
       For x ∈ Go' do
          p1(x) = Selector (x, GN')
       end
end
```

- Improve the initial isomorphism p_1 to p_i iteratively. If we have a isomorphism p_{n-1}, we can construct p_n as follows:

```
S = {x ∈ Go | pn-1(x)!= <Empty>}
For x ∈ S do
    P = Parent(x)
    K = Parent (pn-1(x));
    For y ∈  P do
        If  Selector  (y,  K)  !=
        <Empty Set> ; then
           pn(y) = Selector (y, K)
        end
    end
end
```

4.1.4 **Overcoming Instruction Reordering**

Instruction reordering challenge that occurs due to pipelining / compiler optimization can be overcome by using ***small primes product***. The problem is essentially that two blocks are isomorphic but they differ in the order of

instructions. To detect this, we assign each instruction in the instruction set a unique small prime number.

We compute the product of the prime numbers that correspond to each instruction in a block 1 and compare it to the product of primes corresponding to instructions in block 2. If the product is same, we know that the blocks are same (isomorphic).

4.2 Complexity

The complexity of the algorithm is O(nm) where n is the number of nodes in the graph of executable Old and m is the number of nodes in the graph of executable new.

Since, we are in the worst possible scenario, comparing each node (which is itself a graph) of Old to all the nodes of New to come to a conclusion that a corresponding mapping exists, there can be a maximum number of n*m comparisons.

Essentially, this is quadratic time ~O(n*n) with a slight caveat of implementation specific optimizations.

5. Alternative Approaches

5.1 Entity Name Matching

Description: The code matching techniques matches the code elements like functions, file names and data members. These are referred to as immutable entities and the matching can happen at a certain tuple level where the (class name, function name) might be matched. [12]

Complexity: Such algorithms' complexities can be linear to the size of the code entities if the matching is instruction order specific; otherwise the implementations are usually of the order O (nm) (where n and m are the size of the entities that are being matched). This depends on how the entities are formulated as well and what is the flexibility allowed with respect to a change

Advantages: Simple to implement and use. Differences of patterns (than just plain differences) can be obtained by combining this technique with effective data mining techniques.

Drawbacks: The code changes cannot effectively translate to the amount of logical changes going into the program flow. Thousands of lines of codes can be added and deleted without really affecting program flow; hence, it is not a very useful metric in quantifying the nature of the *logical* change in a binary. Does not abstract the syntax at any level, thus indirect changes are always present in the results

5.2 Syntax Tree Matching

Description: The algorithm [6], which is based on a dynamic programming scheme, is also used in pointing out differences between two programs. This approach involves the creation of syntax trees of the various tokens. The algorithm aims at matching the trees and pointing out the differences. [7]

Complexity: The time complexity of the algorithms belonging to this genre are *O* (Tree1Nodes, Tree2Nodes), where Tree1Nodes and Tree2Nodes are the numbers of nodes of the trees, respectively. The space requirement is the space requirement is *O* (Tree1Nodes +

Tree2Nodes). This is the implementation of Yang in [6].

Advantages: It can be used to determine Dynamic Software Updating metrics [7] in which a tabulation and summarization of simple changes to successive versions of programs is needed which is partially achieved by matching the abstract syntax tree representation of the programs. Good heuristics can reduce the nodes that are being matched, making the algorithm efficient. Syntax abstraction helps in removing indirect noise

Disadvantages: Grammar specific and hence not predicting the accuracy of binary code change. Too sensitive to small changes as nested control structures must map to every level. Implementations are also more focused on the syntax rather than the semantics.

5.3 BMAT

BMAT [BMAT] is considered one of the optimum and fairly effective method of detecting changes in binaries without the knowledge of the source code changes. It is extensively used in profile based propagation, which ahs applications in binary instrumentation and code coverage.

BMAT uses a three tiered approach towards acknowledging binary changes. The figure 1 shows an overview of the whole matching process.

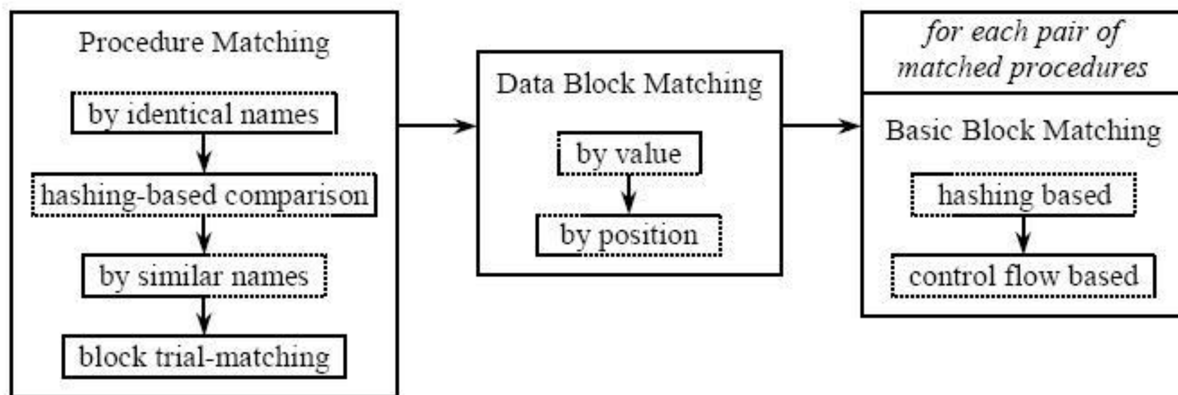

Figure 1. Overview of the BMAT matching Process [1]

The procedures are matched first by basic and extended compiler generated names and then the nature of differences are calculated by the difference in their hash values, which are computed based on the building blocks of the procedure: branches and data values.

In Procedure Mapping, BMAT maps Procedure names based on their basic and extended (based on compiler settings) names. For matching the procedures with different names, a fuzzy match is performed based on block trial matching. A certain heuristic is maintained through which the decision is taken whether a block was a successful match or not. This trial-matching is a simplified, one-pass version of the hashing-based basic block matching algorithm performed later within each pair of matched procedures.

In data block matching, we also use a hashing-based algorithm to match data blocks in the

two binaries. Finally, in the basic block matching we match the code contents using multiple hash passes with different level of fuzziness.

Complexity: O (n^2) as each functional unit (n) is compared against the other

Advantages: Binary Change is being isolated which is indifferent to source code changes. Fuzziness level can be defined according to the nature of differences.

Disadvantages: Cost of running is high. Small offsets caused by variables, stack shifts and register renaming causes the hash comparisons to fail if the hash function is not chosen appropriately.

6. Key Open problems/challenges

6.1 Logical Editing

The matching problem can be easily solved, if we can capture the logical delta between binaries rather than the state delta. Unfortunately, the existing tools only allow us to capture the state change rather than the logical change. [8]

6.2 Hybrid Matching

A combination of all existing matching techniques can complement each of the matching techniques as none is perfect. A technique can be used that can run all matching queries based on disparate algorithms and then merge the common results.

6.3 Dynamic Information Usage

Dynamic Invariant can help in detection of code paths that potentially impact the program if altered. The example of dynamic invariants may be useful in identifying the matches of variable values at the entry and the exit of the function. This can effectively help in profiling and detecting logical changes to program logic rather than just plain code. [9]

7. Conclusion

We can conclude by presenting one of the recent implementations of Code Churn quantification.

Binary Code change quantification can be used come up with metrics that can be used in Defect Prediction in a software life cycle. Metrics such as Block Coverage, Arc Coverage, Frequency of Churn, Number of Developers etc. can be used to get the multiple regression equation of the form:

$$\mathbf{Y = c + a_1PC_1 + a_2PC_2 + \ldots + a_nPC_n,}$$

where a_1, a_2 …., a_n are regression coefficients and PC_1,PC_2 …., PC_n are the principal components and **Y** is the estimated value of post release failures. [10]
This is one of the recent examples of the implementations of quantifying binary churn.

8. References

[1] Z. Wang, K. Pierce, and S. McFarling. BMAT - a binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism*, 2, 2000

[2] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. ACM TOSEM, 6(2):173–210, 1997.

[3] Halvar Flake. Structural comparison of executable objects. In DIMVA, pages 161–173, 2004.

[4] Graph-based comparison of Executable Objects
Thomas Dullien, Ruhr-Universitaet Bochum
Rolf Rolles, University of Technology in Florida
{*thomas.dullien, rolf.rolles*}*@sabre-security.com*

[5] DataRescue. IDA Pro disassembler
http://www.datarescue.com/idabase.

[6] W. Yang. Identifying syntactic differences between two programs. Software - Practice and Experience, 21(7):739–755, 1991.

[7] (I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In MSR'05, pages 2–6.)
[8] [E. Lippe and N. van Oosterom. Operation-based merging. In SDE'92, pages 78–87, 1992.]

[9] M. D. Ernst. Dynamically Discovering Likely Program Invariants. Ph.D. Disseratation, University of Washington, Seattle, Washington, Aug. 2000

[10] Nachiappan Nagappan, Thomas Ball, Brendan Murphy, Using Historical In-Process and Product Metrics for Early Estimation of Software Failures , Microsoft Research, Redmond, WA

[11] Alfred V. Aho, J. D. U., Ravi Sethi: Compilers: Principles, Techniques, and Tools, Addison Wesley

[12] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.