Graph-based comparison of Executable Objects (English Version)

Thomas Dullien[1] and Rolf Rolles[2]

[1] Ruhr-Universitaet Bochum
thomas.dullien@sabre-security.com
[2] University of Technology in Florida
rolf.rolles@sabre-security.com

Résumé A method to construct an optimal isomorphism between the sets of instructions, sets of basic blocks and sets of functions in two differing but similar executables is presented. This isomorphism can be used for porting recovered information between different disassemblies, recover changes made by security updates and detect code theft.

The most interesting applications in the realm of security are in malware analysis where the analysis of a family of trojans or viruses can be reduced to analyzing the differences between the variants, and in recovering the details of fixed vulnerabilities when the vendor of the security patch refuses to disclose details.

A framework implementing the described methods is presented, along with empirical data about it's performance when analyzing multiple variants of the same malware and recovering vulnerability details from security updates.

1 Introduction

While programs that compare different versions of the same source code file have been in widespread use for many years, very little focus has so far been placed on the importance of detecting and analyzing changes between two versions of the same executable.

Without an automated way of detecting source code changes in the object code and porting analysis results between disassemblies of related executables, the party prompted with analyzing the changes is at a disadvantage : Each "round" of reverse engineering requires a massive duplication of work done in prior "rounds" of analysis, while very little work is required to change the source code and recompile.

Both malware authors of high-level-language virus families such as SoBig and software vendors try to exploit this asymmetry : Both want to buy time, one side to allow customers to patch software before attackers can build automated attack tools, the other side to allow their malware to propagate before detection signatures are available.

This paper presents a novel approach for solving this : Given two executables A and A', a bijective mapping between the functions in A and A' is constructed

by iteratively improving a partial graph isomorphism on the call-graphs of the executables. Once that mapping is generated, a bijective mapping between the basic blocks of a pair of functions f, f' is constructed by using the same iterative improvement of a partial graph isomorphism on the flow-graph of the functions. Finally, given a pair of two corresponding basic blocks β, β' an isomorphism of the instructions is generated by treating the sequence of instructions as a special graph and proceeding as above.

2 Previous Work

Automatically analyzing and classifying changes to source code have been studied extensively in literature before, and listing all relevant papers seems to be out of scope for this paper. Most of this research focuses on treating the source code as a sequence of lines, and applying a sequence-comparison algorithm [11][12].

The problem of matching functions in two executables to form pairs has been studied in [3,**?**], although focused on reuse of profiling information which allowed the assumption of symbols for both executables being available. Other work has been done with focus on efficient distribution of binary patches [6] [8]. Both approaches, while finding differences between two binaries, are incapable of dealing with aggressive link-time profiling-information-based optimizations and will generate a lot of superfluous information in case register allocation or instruction ordering has changed. A bytecode-centric approach to find sections of similar JAVA-code is studied in [9].

Another approach to binary comparison also dealing with graph isomorphisms was discussed in [13] : Starting from the entry points of an executable basic blocks are matched one-to-one based on instructions present in them. If no matching is possible, a change must have occurred. Due to the reliance on comparing actual instructions, a significant number of locations is falsely identified as changed - the paper mentions that about 3-5 % of all instructions change between two versions of the same executable. Furthermore, the discussed algorithm seems to match weakly in situations where the call-graph has a low connectivity or significant changes in the order of instructions are present.

The work presented in this paper is a direct extension of the methods and code presented in [10].

3 Structural Code Analysis

Comparing different variants of the same executable (or just two arbitrary executables that share a significant amount of code) has to deal with the problem of the same source code being compiled to conceivably very different representations on the assembly level. A number of common changes that occur between two variants of the same executable are :

1. *Different Register Allocation*

Depending on the compiler's optimization settings and the changes in the code, different registers will be assigned to identical instructions.

2. *Instruction Reordering*
   Depending on the compiler's modelling of the pipelining of the CPU, individual instructions will be reordered.
3. *Branch Inversion*
   In many situations, the compiler will attempt to optimize the alignment of basic blocks by inverting the condition of a branch and exchanging the two basic blocks to which this branch could lead.

Obviously, significantly more severe changes can occur. The main observation on which the methods presented in this paper is built is that the callgraph of an executable stays largely the same[3], even if compiled with a different compiler and for a different architecture.

Instead of focusing on the concrete assembly level instructions obtained via disassembly, the focus of the presented approach are the structural properties of the executable, specifically the basic abstraction of functions and basic blocks as well as their relation to each other.

3.1 Notation

This paper will use quite a few terms from graph theory, thus a few notations need explaining :

The notation $\mathfrak{P}(S)$ means the power set of a given set S.

Whenever the word "graph" is used in this paper, it refers to a possibly cyclic directed graph consisting of a set of *nodes* and a set of *edges*. A simple capital letter is used to denote a graph, and the superscripts to the letter are used if either the set of nodes or edges is referred to.

Thus graph G consists of the set of nodes $G^n := \{G_1^n, \ldots, G_m^n\}$ and the set of edges $G^e := \{G_1^e, \ldots, G_k^e | G_i^e \in G^n \times G^n\}$.

For later use, we define the functions

$$\begin{aligned} \text{up} &: G^n \to \mathfrak{P}(G^n) \\ \text{down} &: G^n \to \mathfrak{P}(G^n) \end{aligned}$$

which map a given node G_i^n to the subset of G^n that are direct "parents" of G_i^n respective to the subset of G^n that are direct "children" of G_i^n.

3.2 An executable as Graph of Graphs

We treat the executable as a *graph of graphs*. This means that the executable is viewed as a multi-edged directed graph A which has all functions retrieved

[3] It is known that some modern compilers can change the callgraph significantly through inlining even complex functions. Studying the applicability of the presented methods needs to be done once code generated by these compilers becomes more widely used

from the disassembly as nodes and the call relations between these functions as edges.

Every node $A_i^n \in A^n$ is a graph itself, with it's nodes consisting of the individual basic blocks in the disassembly and the edges representing their branch relations. Such graphs are usually called *control flow graphs* or short *cfg*.

Each basic block itself (this means each node in the graph represented by A_i^n) is a graph as well, albeit of very simple form : A sequence of assembly-level instructions.

3.3 Retrieving the information

In order to retrieve these graphs from an executable, a good disassembly of the binary is needed. The industry standard for disassembly is [7], mainly due to its excellent cross-platform capabilities coupled with a programming interface that allows retrieval of the needed information without knowledge of the underlying CPU or its assembly. This facilitates implementing the described algorithms only once but testing them on executables built for different architectures.

Indirect calls and disassembly problems In many cases creating a complete call-graph (which represents all possible relations between the different functions) from a binary is not trivial. Specifically indirect subfunction calls through tables (very common for example in C++ code that uses virtual methods) are hard to resolve statically.

In the presented approach, such indirect calls whose targets cannot be resolved statically, are simply ignored and treated as a regular assembly-level instruction. In practice, this does not yield many problems. The big risk is to have non-connected sections of the call-graph in which not a single fixedpoint was generated which would lead to that subsection of the graph not being properly matched. Due to the many different properties that can be used to generate fixedpoints (see Section 4), this is not a problem in practice.

4 Structural Matching

The general idea of the presented approach is the following : Given two executables, the graphs A and B are constructed. Then a number of "fixedpoints" in the two graphs are created : Two elements (one each from A^n and B^n are searched that can be easily determined to represent the same item in both executables.

These fixedpoints are used for creating more fixedpoints iteratively until the mapping can no longer be improved.

Once we have matched the maximum number of functions, we can match basic blocks in the same manner. Since we already have an isomorphism that allows us to retrieve two associated functions, we just have to match the nodes

of two *cfg*'s by identifying fixedpoints and using that information for matching more and more nodes.

Once we are down to the basic block level, we can treat two matching basic blocks as graph (of a very simple form) again, and construct an isomorphism in much the same manner.

4.1 Selectors

A *Selector* is essentially just a mapping that, given a node $A_i^n \in A^n$ of a graph and a set of nodes in another graph returns either one element from the given set or the empty set, e.g.

$$s : A^n \times \mathfrak{P}(B^n) \rightarrow B^n \cup \emptyset$$

The selector's job is to select a single node from a set of given nodes that is most "similar" to A_i^n, or, if more than one candidates with the same "similarity" exists, to select nothing at all.

It is intuitively clear that the probability of a selector returning an empty set rises with larger input sets.

4.2 Properties

A *Property* π is defined as a mapping that maps two graphs A and B to subsets of their node sets :

$$\pi(A, B) \rightarrow (A'^n, B'^n) \text{ with } A'^n \subset A^n \text{ and } B'^n \subset B^n$$

The purpose of such a mapping is reducing the size of the sets used by a selector in order to improve the probability for the selector to return a non-empty result.

4.3 Graph Isomorphism via fixedpoints and propagation

Generating fixedpoints Given a selector s, an approximate graph isomorphism $p : A^n \rightarrow B^n$ can be constructed by constructing an initial isomorphism p_1 and then using this to construct improved versions until one reaches a result that can not be further improved.

The initial isomorphism $p_1 : A^n \rightarrow B^n$ is constructed by simply defining $p_1(x) \rightarrow s(x, B^n)$.

This simple construction can be significantly improved if a number of properties are available. Let $\Pi = \{\pi_1, \ldots, \pi_j\}$ be a set of properties. An improved initial isomorphism would be constructed as follows :

for $\pi \in \Pi$ **do**
  $(K, L) \leftarrow \pi(A, B)$;
  for $x \in K$ **do**
    define $p_1(x) \rightarrow s(x, L)$
  end
end

Propagation of fixedpoints Given the initial mapping p_1, further improved isomorphisms p_i can now be constructed iteratively in the following manner :

Input : p_{n-1}, s, A, B
Result : p_n
$S \leftarrow \{x \in A^n | p_{n-1}(x) \neq \emptyset\}$;
for $x \in S$ S **do**
  P $\leftarrow$ `up` (x);
  K $\leftarrow$ `up` ($p_{n-1}(x)$);
  for $y \in P$ **do**
    if $s(y, K) \neq \emptyset$ **then**
      define $p_n(y) \rightarrow s(y, K)$
    end
  end
end

In plain words the above algorithm retrieves nodes for which p_{n-1} has a useful mapping, and then examines only the sets nodes that are direct "parents" of a node and it's image under p_{n-1}. Since these sets are significantly smaller than the sets examined beforehand, the odds for s returning a non-empty result are enhanced.

The above algorithm can clearly be run with *down* instead of *up*, and best results are achieved by alternating between the two.

4.4 Small Primes Product (SPP)

One of the most common changes between two basic blocks in two executables is a change in instruction ordering. An algorithm to quickly determine if two basic blocks (or even two functions) have the same instructions (but possibly in different order) is therefore of high value – it can be directly used to generate additional initial fixedpoints.

The problem To phrase the problem more concisely :

Let $\mathcal{A} := \{\alpha_1, \ldots, \alpha_m\}$ be an alphabet with m different elements. Let $\mathbb{S}_n$ be the permutation group in n elements. Given two words of length n, say, $\mathfrak{a}, \mathfrak{b} \in \mathcal{A}^n$, one wants to determine if a permutation $\sigma \in \mathbb{S}_n$ exists so that $\sigma(\mathfrak{a}) = \mathfrak{b}$. We will denote the k-th letter of a word $\mathfrak{a}$ by writing $\mathfrak{a}_k$.

A first solution Let $P_m := \{3, \ldots, \rho_m\}$ be the set of the first m odd prime numbers. Furthermore consider the mapping

$$\tau : \mathcal{A} \rightarrow P_m, \quad \tau(\alpha_i) \rightsquigarrow \rho_i$$

which assigns a unique small prime number to each element in the alphabet. We then calculate the product of all letters in $\mathfrak{a}, \mathfrak{b}$ and verify that

$$\prod_{i=1}^{n} \tau(\mathfrak{a}_i) = \prod_{i=1}^{n} \tau(\mathfrak{b}_i)$$

The above condition is equivalent to the existence of a σ with $\sigma(\mathfrak{a}) = \mathfrak{b}$ because of the uniqueness of prime decompositions and the fact that multiplication is commutative.

Adjusting to reality : mod 2^{64} arithmetic Unfortunately large integer arithmetic is rather expensive, and the above method is thus not directly feasible for real-world applications. If we limit all arithmetic above to calculations mod 2^{64}, we can use the normal in-register multiplication of our x86-CPU. This removes the expense of large integer arithmetic at the cost of risking to claim falsely that a σ with $\sigma(\mathfrak{a}) = \mathfrak{b}$ exists.

Quantifying the exact risk is tricky as it depends on the probabilities of the occurrence of a particular α. We can nonetheless calculate an upper boundary for the risk of a false claim under the assumption that all $\alpha \in \mathcal{A}$ occur with identical probability.

For any given word $\mathfrak{c}$ the following inequation holds :

$$\prod_{i=1}^{n} \tau(\mathfrak{c}_i) \leq p_m^n \tag{1}$$

The proposed algorithm will falsely claim that σ with $\sigma(\mathfrak{a}) = \mathfrak{b}$ exists if and only if

$$\prod_{i=1}^{n} \tau(\mathfrak{a}_i) = k2^{64} + c \tag{2}$$

$$\prod_{i=1}^{n} \tau(\mathfrak{b}_i) = j2^{64} + c \tag{3}$$

with $k \neq j, c < 2^{64}, \mathfrak{a} \neq \mathfrak{b}$. We can assume without loss of generality that $k > j$. From the above it becomes evident that there is a maximum of $k - 1$ values of $\prod_{i=1}^{n} \tau(\mathfrak{b})$ which satisfy equation (3). From (1) it follows that

$$k \leq \frac{p_m^n}{2^{64}}$$

The total number l of words $\mathfrak{c}_{(1)}, \ldots, \mathfrak{c}_{(l)} \in \mathcal{A}^n$ for which

$$\prod_{i=1}^{n} \tau(\mathfrak{c}_{(1),1}) \neq \cdots \neq \prod_{i=1}^{n} \tau(\mathfrak{c}_{(l),1})$$

holds is given by $l = \binom{n+m-1}{n}$ as we can model the above product as a simple combination with repetition.

We can thus claim that the odds ρ of two randomly chosen words $\mathfrak{a} \neq \mathfrak{b}$ satisfying

$$\prod_{i=1}^{n} \tau(\mathfrak{a}_i) \equiv \prod_{i=1}^{n} \tau(\mathfrak{b}_i) \mod 2^{64}$$

is smaller or equal to

$$(\frac{p_m^n}{2^{64}} - 1)\binom{n+m-1}{n}^{-1} = (\frac{p_m^n}{2^{64}} - 1)\frac{(m-1)!n!}{(n+m-1)!}$$

One should keep in mind that this is a very rough upper boundary which can be improved significantly. Such improvement would be beyond the scope of this paper.

The important conclusion to draw is that using the proposed method on an alphabet with 100 elements is definitely safe for words that are shorter than 14 elements, and very likely for a significant stretch beyond that.

SPP and code similarity Our implementation uses SPP for identifying sequences of instructions with matching mnemonics. We use the disassembler-assigned index for each mnemonic to index into a table with small primes, and calculate the result as an *unsigned long long*. This is done on both the function and the basic block level.

4.5 Example Selectors and Properties

Generic Properties Many different properties can be thought of. In our example implementation, we have used several different properties for graphs A, B to good effect, with the best results coming from combining all of the below.

All mappings are in the form of

$$\pi_1 : (A^n, B^n) \rightarrow (\{A_i^n, \ldots, A_k^n\}, \{B_j^n, \ldots, B_l^n\})$$

with certain criteria that the A_i^n, B_i^n have to fulfill. For brevity we only list the criteria and a short explanation of their meanings :

1. *k-Indegree Nodes / k-Outdegree Nodes*

   $$\sharp(\mathrm{up}(A_i^n)) = k \text{ and } \sharp(\mathrm{up}(B_i^n)) = k$$

   This means that we select nodes whose indegree is exactly k. Replacing *up* with *down* yields all nodes with outdegree of excatly k. Note that selecting a k of zero will retrieve all root (or alternatively all leaf) nodes.

2. *Recursive Nodes*

   $$A_i^n \in \mathrm{up}(A_i^n) \text{ and } B_i^n \in \mathrm{up}(B_i^n) = 0$$

   This selects nodes that have a link to themselves, selecting only functions that recursively call themselves.

Properties specific to callgraphs Most properties are not specified on abstract graphs but use the underlying assembly code for specifying properties like the following :

1. *Same Name*
   Clearly the most obvious property : Many nodes in the callgraph of an application will have names, either from debug information that is available or because of import/export information in the executable.
2. *Same String Reference*
   Nodes in the callgraph can be selected by common string references, indicating functions that all contain code referring to the same string.
3. *Same SPP*
   Nodes in the callgraph can be selected by common SPP.

Properties specific to CFGs Both the *Same String Reference* and the *Same SPP* property can be directly applied to CFGs. In addition to these two, the following property has shown to be useful :

1. *Same subfunction call*
   Nodes in the CFG may contain calls to subfunctions.
   At the point where CFG-isomorphisms are calculated, a good isomorphism p_c for the callgraph is already available. One can therefore select nodes that call subfunctions that are the same under p_c.

Properties specific to the Instruction-level When matching instructions on the instruction-level-graph (which is in essence just a sequence), the *same string reference* and *same subfunction call* properties are used.

A selector for the callgraph We associate a 3-tuple with each node in the callgraph. This 3-tuple consists of the number of basic blocks in the function, the number of edges linking them to form the CFG, and the number of subfunction calls found in the basic blocks.

The selector for callgraph nodes works simply as follows : The 3-tuples are interpreted as simple vectors in euclidian space, and the euclidian distance between the tuple of each element in the supplied set and the tuple of the supplied element is calculated. If a unique tuple with minimal distance is found, the selector returns the associated node.

More formally :

$$s_c(x, A) := \begin{cases} a \text{ si } \exists a \in A, \forall b \in A, b \neq a, |x - a| < |x - b| \\ \emptyset \text{ sinon} \end{cases}$$

A selector for the CFGs In the case of *cfg*s we work again with 3-tuples of natural numbers. The construction of this selector is based on the observation that for small changes in the function, the changes to the *cfg* are often localized

to a region of the graph. This implies that for a given basic block a, the change will be either *below* or *above* that block. This implies that either the number of basic blocks on the shortest path *to* a or the number of basic blocks on the shortest path *from* a to the end of the function remains constant.

The second observation was that most functions include a significant amount of error checking which is represented in the *cfg* as paths bypassing most of the functions and jumping directly to the exit node.

We thus associate a 3-tuple with each node in the *cfg*. This 3-tuple consists of the number of blocks on the shortest path to a function exit, of the number of blocks on the shortest path from the functions entry point, and the number of subfunction calls made in that basic block.

The disadvantage with this approach is that an inserting a basic block into the *cfg* can skew the signature of all blocks that are dominated by it.

In order to deal with this issue, a special selector is used : It takes a special δ-Parameter. The definition is more or less the same as in the calltree situation then :

$$s_c(x, A, \delta) :=$$
$$\begin{cases} a \text{ si } \exists a \in A, \forall b \in A, b \neq a, |x - (a + \delta)| < |x - (b + \delta)| \\ \emptyset \text{ else} \end{cases}$$

During the propagation of fixedpoints as described in 4.3.2, the δ parameter is calculated by calculating the difference between the two signatures in the fixedpoint.

A selector for the Instruction-Level For building the instruction-level isomorphism, we essentially take the distance to the entry and the distance to the exit of a basic block as signature and apply the same algorithm as described above.

5 Applications

The capability of building an isomorphism down to the instruction level offers many interesting applications.

5.1 Porting comments for analysis of malware variants

For demonstration purposes, we obtained two samples of the Bagle trojan, specifically Bagle.X and Bagle.W. A thorough analysis of the Bagle.W-sample was conducted, with a detailed disassembly in which all functions were properly named and most of the database thoroughly commented.

We then produced an untouched disassembly of Bagle.X : No meaningful function names were present, and the disassembly was completely uncommented.

After running our implementation of the described algorithms on the two disassemblies, all but 6 functions in the untouched disassembly had been successfully associated with their counterpart in the already-analyzed disassembly. Furthermore, only 3 functions had changed in any significant manner.

Out of the 1524 comments in the matched functions, all but 10 were successfully transferred between the disassemblies.

All in all, the task of analyzing the Bagle.X variant was reduced to examining three changed functions and six (very small) unmatched functions. Almost all function names and comments that had been created for the previous database could be re-used. Running our analysis took less than 30 seconds.

5.2 Recovering vulnerability information

H323ASN1.DLL After the NISCC published information about vulnerabilities in multiple H.323 parsers, the question arose where the relevant mistake in Microsofts ISA Server product was. Microsoft refuses to publish detailed information about the vulnerability they fix. According to the NISCC report, the problem was located in ASN.1 decoding.

Both the pre- and post-patch versions of H323ASN1.DLL were analyzed, and a total of 8 changed functions (out of 1655) detected.

The changes could be classified into two cathegories :

1. Introduced sanity checks on untrusted values specifying the number of words to decode from an ASN.1 stream
2. Introduced sanity checks prior to calls to `ASN1PERDecZeroTableCharString-NoAlloc()`

In the second case, a 32-bit integer from the ASN.1 stream is passed on to ASN1PERDecZeroTableCharStringNoAlloc() as second argument. The patched variant introduces a range check to make sure this second argument is smaller than 129.

A closer inspection of ASN1PERDecZeroTableCharStringNoAlloc() reveals that the function calculates the size of memory allocation based on the formerly untrusted value – an attacker was able to set this value in a manner that the calculation would exceed MAXUINT and thus be of very small size. The subsequent copy-operation would then corrupt the heap, allowing an attacker to gain control in the next round of heap consolidation. Instead of fixing the issue at the core (e.g. in the MSASN1.DLL library), a range check was added into the calling application (H323ASN1.DLL).

The update thus disclosed to an examining party that every call to ASN1PERDecZeroTableCharStringNoAlloc() needs to have argument checking done *before* the call is issued. A short system-wide scan was conducted to see if other applications besides ISA Server use the relevant function in dangerous way. Two other instances were found : The Windows-internal H.323 Multimedia Provider Library (which allows arbitrary applications to easily process H.323 data) and Microsoft's Video Conferencing Software Netmeeting. Neither does proper range checking on the function in question.

The result was that the update to H323ASN1.DLL fixed one bug but alerted anyone with the capability to analyze patches to two further remotely exploitable vulnerabilities which were not fixed at the time.

Microsoft was contacted and the issues were fixed a few months later, in MS04-11.

The total analysis took less than 3 hours time, the actual running time of the algorithms less than 5 minutes.

SSL/PCT Parser In April, Microsoft issued an update to SCHANNEL.DLL, the library responsible for handling SSL communication. According to their security bulletin, they removed a security problem that allowed attackers to take full control of any computer running an SSL-based server. No technical details were provided, except that the problem itself lay in a part of the library responsible for parsing PCT packets [4].

More than 20 changed functions were detected in total, but only one with a name that implied it was involved with PCT parsing. An examination of the function Pct1SrvHandleUniHello() revealed that the old version had taken a string, NOT'ed every character and appended it to the original string. The new version was changed in such a manner that it ensured the combined string would not exceed 32 characters.

Detecting and understanding the vulnerability (a vanilla stack-smash with EIP overwrite) took less than 30 minutes. Subsequently, code was constructed to reach the appropriate location in the binary. Within 5 hours, EIP could be overwritten with an arbitrary value, and within 10 hours of the start of the analysis, a program that reliably exploited the vulnerability was created.

6 Summary

It has been shown that nondisclosure of vulnerability information is not a promising deterrent to would-be-attackers and that security updates can be reverse engineered in relatively little time (given the right tools). It has furthermore been shown that special care has to be taken when releasing security updates, as the information in the patch has to be assumed to be public. An incomplete bugfix can do more harm than good by disclosing the existence of other (unfixed) bugs along with the fix.

The presented work furthermore implies that the common practice of leaving one or two weeks between the publication of a security update and installing the patch is highly dangerous.

Leaving the politics of vulnerability disclosure out, it has been shown that analysis of binaries based only on structural properties of the code is a promising field of research, as it allows analysis of executable code without the need to abstract to an intermediate language or CPU-specific analysis engines.

[4] PCT is a legacy-protocol that was obsoleted by TLS and is supported for legacy browsers

Références

1. M. Goossens, F. Mittelbach and A. Samarin, *Der LaTeX-Begleiter*, Addison-Wesley, 1995.
2. A. V. Aho, R. Sethi and J. D. Ullmann, *Compilerbau*, 2nd Edition, Oldenburg Verlag, 1999.
3. Z. Wang, K. Pierce and S. McFarling, *BMAT - A Binary Matching Tool for Stale Profile Propagation*, The Journal of Instruction-Level Parallelism (JILP), Vol. 2, May 2000.
4. Z. Wang, K. Pierce and S. McFarling, *BMAT - A Binary Matching Tool*, 2nd ACM Workshop on Feedback-Directed Optimization, November 1999.
5. Z. Wang Division, *BMAT – A Binary Matching Tool* `http://citeseer.nj.nec.com/262615.html`
6. Pocket Soft Inc., *RTPatch – Software Update Tool*, `http://www.pocketsoft.com/whitepapers/whitepaper.html`
7. DataRescue, *IDA Pro Disassembler*, `http://www.datarescue.com/idabase`
8. B. S. Baker, U. Manber and R. Muth, *Compressing Differences of Executable Code*, ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS), pp. 1–10, 1999, `citeseer.nj.nec.com/baker99compressing.html`
9. B. S. Baker and U. Manber, *Deducing Similarities in Java Sources from Bytecodes*, Proc. of Usenix Annual Technical Conf., pp. 179–190, 1998, `citeseer.nj.nec.com/baker98deducing.html`
10. Halvar Flake, *Structural Comparison of Executable Objects*, DIMVA, pp. 161–173, 2004.
11. D. S. Hirschberg, *Algoritms for the Longest Common Subsequence Problem*, Journ. of the ACM, Vol. 24 Nr 4, pp. 664–675, ACM Press, 1977.
12. J. W. Hunt and T. G. Szymanski, *A fast algorithm for computing longest common susequences*, Commun. ACM, Vol. 20 Nr 5, pp. 350–353, 1977, ACM Press.
13. T. Sabin, *Comparing binaries with graph isomorphisms*, `http://razor.bindview.com/publish/papers/comparing-binaries.html`