An Efficient Method for Finding Semantic Differences in Binaries

Steven Houston: shouston@eecs.berkeley.edu

May 20, 2008

Abstract

An efficient method is developed for finding the semantic differences between two binary files. Prior work has either focused on locating syntactic differences or has given prohibitively slow methods for finding semantic differences. Roughly, two basic blocks of code are semantically equivalent if they give identical outputs when given identical inputs, and two basic blocks are syntactically equivalent if their instructions are identical. We show that the output of a syntactic difference finder can be used as input to a semantic difference finder to significantly reduce the number of basic block comparisons and improve its runtime.

1 Introduction and Background

Binary analysis is the technique of attempting to understand the underlying properties of an executable without using any information from its source code. There are many reasons why the source code may not be available, and unfortunately for most cases of interest to researchers, it isn't. The majority of commercial software vendors do not release source code to protect their propriety algorithms and prevent piracy. Similarly, security analysts rarely find source code pleasantly attached to the malware they wish to analyze.

Additionally, instead of analyzing a single binary executable, it is sometimes of interest to compare two binaries and locate all meaningful differences between them. Such techniques could be used, for example, to discover what effect a particular patch has on an executable.

There currently exist many accurate and efficient tools for finding differences between two binaries (cf. [1], [2], [3]). However, almost all prior work has focused on finding their syntactic differences instead of their semantic differences. Roughly speaking, syntactic differences correspond to any changes in instructions (i.e. the actual syntax of the instructions have changed), whereas semantic differences occur only when input/output behavior changes (i.e. the semantics of the instructions change). More formal definitions of syntactic and semantic equivalences will be given in the next section.

Gao [5] has described a maximum common subgraph isomorphism algorithm that utilizes a theorem prover for finding all semantic differences in binaries. However, standing alone this algorithm is too slow in practice as it tests nearly every pair of basic blocks in the two executables for semantic equivalence. This paper develops a new hybrid technique that seeds Gao's algorithm with results from an efficient syntactic

difference finder, resulting in dramatic improvements in runtime.

2 Definitions and Motivation

A *basic block* is an ordered list of instructions that has a single entry point (the first instruction of the list) and a single exit point (the last instruction of the list). The *variables* of a basic block are the names of all registers and memory locations that are part of any basic block instruction. The *state* at a particular point in time is the current values assigned to the basic block's variables. The *context* of a certain instruction is the state just prior to that instruction executing. We can define the *entry context* of a basic block as the context of its entry point. Similarly, a basic block's *exit context* is defined as the context of its exit point.

The *control flow graph* (CFG) of a function is a directed graph with the basic blocks of the function represented by vertices and program flow between basic blocks represented by directed edges. The *call graph* (CG) of a binary is a directed graph with the functions of the program represented by vertices and program flow between functions represented by directed edges.

Two basic blocks B_1 and B_2 are *semantically equivalent*, denoted $B_1 \sim B_2$, if (1) they always produce identical exit contexts when given identical entry contexts and (2) the exit points have identical instruction types (e.g. both returns or both calls). This definition implies semantically equivalent basic blocks must have the same number of variables. Two blocks B_1 and B_2 are *syntactically equivalent*, denoted $B_1 = B_2$, if they have identical lists of instructions. Note that syntactic equivalence is a much more stringent equivalence relation since $B_1 = B_2$ implies $B_1 \sim B_2$, but the converse is not true.

If initially $B_1 \sim B_2$, then we still have $B_1 \sim B_2$ after the following transformations:

- Consistently renaming any subset of variables (for example, swapping `reg2` and `reg3` at all their occurrences in B_1).
- Reordering instructions when such a reordering does not affect B_1 or B_2's exit context.
- Changing the arguments but not the type of the basic block's last instruction (for example, in basic block reordering).

However, if initially $B_1 = B_2$, then the above transformations will result in $B_1 \neq B_2$.

Compilers often employ these transformations to varying degrees for optimization purposes, so compiling identical source code using two different compilers may produce several syntactic differences with no semantic differences. Thus, when tasked with finding the *meaningful* differences between two binaries, we are motivated to search for the semantic differences rather than the syntactic differences.

3 Problem Formulation

Define a function ϕ over pairs of basic blocks where $\phi(B_1, B_2) = 1$ if $B_1 \sim B_2$ and $\phi(B_1, B_2) = 0$ if $B_1 \not\sim B_2$. This definition will be used in the remainder of the paper. Suppose we develop an algorithm that returns a complete matrix of ϕ values across all pairs of basic blocks. Will this matrix alone be enough to find all meaningful differences in the binaries? Such a matrix would be very sparse, and the over-abundance of 0 entries would prevent us from honing in on the meaningful binary differences. In actuality, we

are only interested in finding the semantic differences between basic blocks "near" each other in the control flow of the program.

This can be formulated into the problem of finding the maximum common subgraph isomorphism between two graphs, where maximum is not defined as the maximum number of matched nodes but instead as the maximum sum of the value of some function on all matched node pairs. For a particular isomorphism instance, we will refer to this function as the *isomorphism metric*. Gao formulated two problems as maximum common subgraph isomorphism problems.

CFG Isomorphism Problem: Finding semantic differences between two functions. The input graphs are set to the functions' control flow graphs, and the isomorphism metric is set to ϕ. As a result, the isomorphically mapped basic blocks with ϕ values of 0 and all unmapped basic blocks correspond to the meaningful semantic differences we are tasked to find.

CG Isomorphism Problem: Finding semantic differences between two binaries. The input graphs are set to the binaries' call graphs, and the isomorphism metric is set to ϕ_f, where $\phi_f(g,h) = \sum \phi(B_1, B_2)/N$ over all mappings $B_1 \rightarrow B_2$ in the CFG isomorphism from g to h, with N defined as the number of basic blocks in g or h, whichever is greater.

Straightforward implementation of this results in solving many CFG isomorphism instances to find a single solution to the CG isomorphism problem. This is precisely what Gao's semantic difference finder does as detailed in the next section.

4 Original Semantic Difference Finder

For every pair of functions (g,h), Gao's semantic difference finder first finds the maximum common subgraph isomorphism of their respective control flow graphs. From this maximum common subgraph, it assigns a matching score ϕ_f to the pair (g,h) equal to the sum of the isomorphism metric ϕ of mapped basic blocks divided by the total number of basic blocks. The isomorphism metric ϕ of two basic blocks is defined as above ($\phi(B_1, B_2) = 1$ if $B_1 \sim B_2$ and $\phi(B_1, B_2) = 0$ otherwise) and is calculated by calling the theorem prover STP on candidate block pairs.

After obtaining $\phi_f(g,h)$ for all pairs of functions, it calculates the maximum common subgraph isomorphism of the two call graphs, using the previously found function pair matching scores ϕ_f as the new isomorphism metric.

To deal with the NP-completeness of maximum common subgraph isomorphism, a branch-and-bound algorithm is utilized [6]. Two lists are kept, a list M with all current matchings and a list P with all potential matchings for nodes not already matched. The high-level pseudocode of Gao's algorithm is given below:

```
Isomorphism (M,P) :
   if Extendable (M,P) then
      v = PickAny(P)
      Z = GetPossibleMatchings(v,P)
      for all w in Z do
         M' = M + (v,w)
         P' = Refine(v,w,P)
         Isomorphism(M',P')
      end for
      P' = Refine(v,null,P)
      Isomorphism(M,P')
```

The `Extendable` function is the bound portion of the branch-and-bound technique. It quickly determines whether adding some (unknown) subset of matchings from P to M could possibly beat the current optimal isomorphism metric sum. The `PickAny` function returns the node that has the highest matching strength and graph connectivity with unmatched nodes. This is a heuristic intended to get the algorithm to converge on the optimal solution as quickly as possible. Finally, the `Refine` function removes the (v, w) matching from P and also removes any other matchings that would isomorphically conflict with the new (v, w) matching.

See Gao's technical report [5] for further details of the algorithm and examples of semantic differences in binaries discovered by his system.

5 Semantic Optimizations

Let V_1 be the array of variables in B_1 in the order they first appear in the block, with variables in the same instructions being processed from right (evaluation side) to left (assignment side). Similarly, let V_2 be the array of B_2 variables in the order they first appear. Suppose $B_1 \sim B_2$. Since the definition of $B_1 \sim B_2$ implies that B_1 and B_2 have the same number of variables, there must exist a permutation $V_1 \rightarrow V_2$ defining the equivalence mapping between the two blocks. The bottleneck of the original semantic difference finder is the high number of calls to the STP theorem prover, which is repeatedly used to find this permutation.

Thus, an immediate goal is to optimize the search for this permutation. As shown in Table 1 above, all experiments indicate that this permutation has a very high tendency to be the identity matrix. In fact, for our test cases, it was the identity matrix 100% of the time. Thus, a strong heuristic when determining the permutation $V_1 \rightarrow V_2$ is to check the identity matrix first. If that fails, all other possible mappings are still checked to ensure soundness. This heuristic resulted in a dramatic decrease in the total number of STP queries (proportional to the total runtime), as shown in Table 2.

Table 1: Diagonal Optimization Statistics

File	$\sim$ Count	% Identity
Aspnet_filter.dll	615	100%
Aspnet_filter.dll[1]	2	100%
Pngfilt.dll[1]	25	100%

Table 2: Diagonal Optimization Speedup

File	# STP Calls Before	# STP Calls After	% Diff
Aspnet_filter.dll	159,593	38,574	76%
Aspnet_filter.dll[1]	5,927	1,996	66%
Pngfilt.dll[1]	7,861	2,836	64%

Suppose $B_1 \sim B_2$ and we have already found the permutation $P_B : V_1 \rightarrow V_2$. Suppose we now wish to find a permutation P_C such that $C_1 \sim C_2$. If B_1 and C_1 have some subset of common variables, can we always reuse some of the P_B mapping? Unfortunately, no. The permutations do not have to be consistent across a control flow graph. For example, consider the following code snippets:

```
Function1:
   input r1;
   r2 = 1 + r1;
   r3 = 2 + r2;
   r2 = foo(r3);
```

[1]Semantic/syntactic hybridization was also used for these particular test cases.

```
   r3 = 4 + r2;
   r4 = 5 + r3;
   return r4

Function2:
   input s2;
   s3 = 1 + s2;
   s4 = 2 + s3;
   s4 = foo(s4);
   s5 = 4 + s4;
   s6 = 5 + s5;
   return s6
```

Each function has two basic blocks, and the blocks from `Function1` are semantically equivalent to the blocks from `Function2`. The permutation for the first pair of basic blocks is $(r_1, r_2, r_3) \rightarrow (s_2, s_3, s_4)$, but the permutation for the second pair is $(r_2, r_3, r_4) \rightarrow (s_4, s_5, s_6)$. Therefore, we cannot rule out possible permutations by simply looking at previously found permutations.

Previously found mappings do establish a good heuristic, i.e., using a previous mapping typically works. However, the diagonal optimization provides an optimal heuristic (it found 100% of the semantic matchings in our experiments) and does not have the overhead of storing and searching through previous mappings. Therefore, the optimization of remembering variable mappings across basic blocks was removed in the final system.

Additional, somewhat non-interesting, optimizations were also added to improve the runtime of the semantic difference finder. For example, hash tables were employed instead of lists for data structures that often needed dictionary-like lookups, redundantly stored data was merged into a single location, etc.

6 Semantic/Syntactic Hybridization

A key point to gather from the previous sections is that the original semantic difference finder solves many CFG isomorphism instances to find a single solution to the CG isomorphism problem. There exist many tools for finding the syntactic differences between binary files very quickly. The crux of this paper is that we can utilize the output of such tools to give our semantic difference finder a "head start" and solve a much smaller subset of CFG isomorphism problems.

We assume the syntactic difference finder reports a matching value m_f for all possible function pairs (g, h), with $m_f(g, h) = 1$ implying all of the basic blocks in g and h are syntactically equivalent and isomorphically mapped to each other, $m_f(g, h) = 0$ implying none of the basic blocks in g and h are isomorphically mapped to one another by the syntactic difference finder, and $0 < m_f(g, h) < 1$ implying some fraction of the basic blocks in g and h are syntactically equivalent and isomorphically mapped by the syntactic difference finder.

For the top-level call graph isomorphism, we partition all possible functions pairs (g, h) into four sets:

- S_1: all (g, h) such that $m_f(g, h) = 1$.
- S_2: all (g, h) such that $0 < m_f(g, h) < 1$.
- S_3: all (g, h) such that $m_f(g, h) = 0$ and neither g nor h is part of a pair in S_1.
- S_4: all (g, h) such that $m_f(g, h) = 0$ and at least one of g and h belongs to a pair in S_1.

Since $\phi_f(g, h) \geq m_f(g, h)$, we set $\phi_f(g, h) = 1$ for all function pairs in S_1. For all function

pairs in S_2 and S_3, we calculate the semantic isomorphism of their CFGs (as described below). For all function pairs in S_4, we set $\phi_f(g,h) = 0$.

This last assignment is based on the following idea: if two functions j and k are syntactically equivalent and "near" each other in the call graph as determined by the syntactic difference finder (i.e. $m_f(j,k) = 1$), then necessarily $\phi_f(j,k) = 1$. Therefore, they should also be mapped to each other by the semantic difference finder since their isomorphism metric is maximized.

Next, when calculating the CFG isomorphisms of two functions, we use the imported syntactic block matching data in a similar manner. We assume the syntactic difference finder reports a matching value m for all possible basic block pairs (B_1, B_2), with $m(B_1, B_2) = 1$ if B_1 and B_2 are syntactically equivalent and isomorphically matched, and $m(B_1, B_2) = 0$ otherwise. We partition all possible basic block pairs (B_1, B_2) into three sets:

- T_1: all (B_1, B_2) such that $m(B_1, B_2) = 1$.
- T_2: all (B_1, B_2) such that $m(B_1, B_2) = 0$ and neither B_1 nor B_2 is part of a pair in T_1.
- T_3: all (B_1, B_2) such that $m(B_1, B_2) = 0$ and at least one of B_1 and B_2 belongs to a pair in T_1.

Since $\phi(B_1, B_2) \geq m(B_1, B_2)$, we set $\phi(B_1, B_2) = 1$ for all basic block pairs in T_1. For all basic block pairs in T_2, we ask the theorem prover whether or not B_1 and B_2 are semantically equivalent. For all basic block pairs in T_3, we set $\phi(B_1, B_2) = 0$.

This last assignment is justified by the following provable property:

> If $m(B, C) = 1$, then $\phi(B, C) + \phi(X, Y) \geq \phi(B, X) + \phi(C, Y)$ for any other blocks X and Y.

In other words, if B and C are syntactically matched, then we cannot do any better in the semantic isomorphism by matching them with other blocks. Thus, $\phi(B_1, B_2)$ can safely be set to 0 whenever B_1 or B_2 are syntactically matched with other blocks.

The heuristics for handling the function pairs in S_4 and the basic block pairs in T_3 drastically reduce the overall number of basic block comparisons. In fact, the theorem prover is only called for basic block pairs in T_2 whose parent functions are in S_2 or S_3.

7 Implementation and System Architecture

The semantic difference finder is implemented using the OCaml functional language. Due to the complexities of the x86 instruction set, we use VinE [8] to disassemble the binary and convert it to a more simplified intermediate representation, called the VinE IR. As described in [5], this simplification could result in a loss of precision, but dealing with that issue is beyond the scope of this project. The CFG isomorphism algorithm is then performed on basic blocks pairs (B_1, B_2) of IR instructions. The theorem prover used to determine whether $B_1 \sim B_2$ is STP [4]. No current cases have resulted in unreturned calls to STP; therefore, there is no STP timeout implemented in the current system.

The syntactic difference finder used is eEye's DarunGrim [2]. It reads an sqlite database file produced by IDAPro [7] and writes the syntactic matching information back to that file. OCaml's

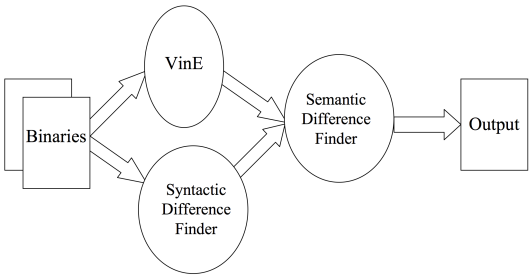

Figure 1: Architecture of the hybrid syntactic/semantic system.

squlite module then interfaces with this database and imports the syntactic matching information into the semantic difference finder. See Figure 1 for an overview of the system architecture.

8 Performance and Quality of Hybridization

The speedup of the hybridization is dramatic, as shown in Table 3 for two different test cases. Each test case found the semantic differences between an unpatched and patched library, and the hybrid runtime was less than 5% of the non-hybrid runtime in both cases. Furthermore, the absolute runtimes are shown in Table 4 for four different test cases run on an Intel Core 2 Duo 6600 (2.4 GHz) with 4 GB of memory.

Additionally, the quality of the subgraph isomorphism matching is not compromised. As shown in Table 5, the overall semantic matching score of the binaries' call graphs actually increases with hybridization. This somewhat surprising result is likely caused by the isomorphism algorithm timeout (since it is NP-complete, a timeout is used to avoid getting stuck on one particular isomorphism). Thus, even though two functions may be semantically equivalent, the CFG isomorphism may reach its timeout before the optimal matching can be found. A timeout is less likely to happen with the hybrid model, since fewer basic blocks are compared resulting in a faster runtime.

Table 3: Hybridization's Reduction of STP Queries

File	# STP Calls, No Hybrid	# STP Calls, With Hybrid	% Diff
Aspnet_filter.dll	38,574	1,996	95%
Pngfilt.dll	77,755	2,836	96%

Table 4: Absolute Hybridization Speedup

File	Runtime, No Hybrid	Runtime, With Hybrid
Aspnet_filter.dll	41 min	1.4 min
Pngfilt.dll	130 min	0.55 min
Umpnpmgr.dll	> 4 hr[2]	15 min
Gdi32.dll	> 4 hr[2]	48 min

9 Related Work

As mentioned above, there is a plethora of literature and tools available for quickly finding syntactic differences in binary executables (cf. [1], [2], [3]). Although these tools are very efficient, they may output non-meaningful differences. Gao [5] described a new technique for finding semantic differences, but his method was prohibitively slow for most input files. This paper takes the best from each of these research streams. By combining the speed of syntactic

[2]The test was halted after 4 hours due to time constraints.

Table 5: Quality of Hybridization

File	Semantic Matching Score, No Hybrid	Semantic Matching Score, With Hybrid
Aspnet_filter.dll	0.980254	0.996474
Pngfilt.dll	0.974620	0.988670

difference finders with the quality of the semantic difference finders, an efficient semantic difference finder is created.

10 Future Work and Limitations

The current platform is not currently implemented to handle inlined functions. However, this could easily be implemented by merging all control flows graphs and the call graph into a single control flow graph representing the control flow of all basic blocks in the executable. A single subgraph isomorphism could then be performed on the entire graph, utilizing the basic block matching scores (and ignoring the function matchings) from the syntactic difference finder. An advantage of the hybridization model presented is that calculating the maximum common subgraph isomorphism on this merged graph will no longer be prohibitively slow.

Furthermore, we noticed that most semantic differences reported by Debin's semantic difference finder were not reported as syntactic differences by DarunGrim. This is opposite the result we want! Upon further investigation, this appears to be caused by the Vine module not handling pointers correctly in some cases. For example, suppose two basic block variables are pointers p_1 and p_2, with $\&p_1 \neq \&p_2$ and $p_1 = p_2$. In some instances with pointers, Vine used the pointer address (the location of the pointer on the stack), instead of the pointer value (the address the pointer is pointing to), thus reporting semantic differences that did not actually exist. The syntactic difference finder, on the other hand, handled the pointers as expected. Correcting the underlying translation into VinE IR was beyond the scope of this project.

Another approach to speeding up a semantic difference finder is to use static/dynamic hybridization instead. One can utilize the information in a dynamically created program trace to collapse functions into single basic blocks, thereby allowing a fast run of the top-level call graph isomorphism. The CFG isomorphism algorithm can then be run on the resulting matched functions only. The author's first attempt at speeding up the semantic difference finder used this method; however, the traces obtained were not of a manageable size for most test cases.

Finally, future work is needed to rigorously justify the soundness of the heuristics used in the hybridization. Specifically, are there any degenerate classes of call graphs and control flow graphs for which the greedy handling of function pairs in S_4 leads to a sub-optimal isomorphism?

11 Conclusions

We have shown that the syntactic difference finder gives an excellent initialization for the semantic difference finder both in terms of quality and performance. This allows the algorithm to operate in a top-down fashion, only performing the initial CFG isomorphisms on likely function matchings. Previous work performed CFG isomorphisms for the vast majority of function pairs, and thus in a bottom up manner called

the theorem prover on nearly every possible pair of basic blocks in the entire binary.

In the test cases, semantic differences were found very quickly, over 95% faster than the previous method, actually resulting an increase in accuracy. Furthermore, the dramatic performance increase enables the algorithm to easily handle inlined functions with a slight change in its internal graph structure.

References

[1] T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *Proceedings of SSTIC*, 2005.

[2] eEye Digital Security. eEye binary diffing suite. http://research.eeye.com/html/tools/RT20060801-1.html.

[3] H. Flake. Structural comparisons of executable objects. In *Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, 2004.

[4] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, 2007.

[5] D. Gao. Technical report: Automatically finding semantic differences in binary executables.

[6] E. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Software - Practice and Experience*, 34, 2004.

[7] Hex-Rays SA. IDAPro. http://www.hex-rays.com/idapro/.

[8] Vine: The bitblaze static analysis component. http://bitblaze.cs.berkeley.edu/vine.html.