BinHunt: Automatically Finding Semantic Differences in Binary Programs

Debin Gao[1], Michael K. Reiter[2], and Dawn Song[3]

[1] Singapore Management University
dbgao@smu.edu.sg
[2] University of North Carolina at Chapel Hill
reiter@cs.unc.edu
[3] University of California, Berkeley
dawnsong@cs.berkeley.edu

Abstract. We introduce BinHunt, a novel technique for finding semantic differences in binary programs. Semantic differences between two binary files contrast with syntactic differences in that semantic differences correspond to changes in the program functionality. Semantic differences are difficult to find because of the noise from syntactic differences caused by, e.g., different register allocation and basic block re-ordering. BinHunt bases its analysis on the control flow of the programs using a new graph isomorphism technique, symbolic execution, and theorem proving. We implement a system based on BinHunt and demonstrate the application of the system with three case studies in which BinHunt manages to identify the semantic differences between an executable and its patched version, revealing the vulnerability that the patch eliminates.

1 Introduction

Many software vendors make the source code of their programs unavailable. When a program needs to be updated (for patching vulnerabilities and errors), they release a new version in binary format but refuse to disclose details of the changes made. However, it is of great interest for consumers of the software to understand the differences in two versions of the program. Binary difference analysis is one of the most useful techniques in finding these differences.

There are many reasons why consumers want to know the differences between the two binary files. For example, when an update of a program is available, the consumers need to make a decision whether to apply the update. Such a decision may require security analysis of the updated version, which is usually an expensive and time-consuming process. Binary difference analysis can simplify and speed up this process because one can reuse the security properties of the earlier version and focus the analysis on the difference between the two versions. A similar application of binary difference analysis is profile reuse in application development [15]. A large program, with millions of lines of code, may be used in a large variety of ways by different users on different machines. To characterize the program behavior, extensive collection of profile data is required. For example, BMAT is a tool that matches two versions of a binary program to propagate profile information from an older, extensively profiled build to a newer build. Another reason why binary difference analysis is useful is that in many cases the differences in two versions of a program correspond to vulnerabilities in the earlier version that the later version patches. One may be able to find vulnerabilities in a program using binary difference analysis, and subsequently exploit those vulnerabilities and attack consumers who have not applied the update.

Although such binary difference analysis is very useful, it is different from the binary difference tools that we use to produce and apply patches (bsdiff, bspatch, xdelta, jdiff, jpatch, and etc.), because many *syntactic differences* may not correspond to *semantic changes* in the program. In the next section, we will carefully define what syntactic and semantic differences are, but intuitively semantic differences correspond to changes in the program functionality, and are what we seek to identify in this paper.

Finding the semantic differences between two binary files is challenging for many reasons. For example, a small change in the source code may cause the compiler to use a different register allocation in other parts of the program in which the corresponding source code remains the same. Similarly, a small change in the source code may change the size of a small number of basic blocks, which further triggers the compiler to re-order many other basic blocks in the binary file. For these and other reasons, a small change in source code may lead to many changes throughout the binary file. For example, in one of the case studies we report, the patch of the `gzip` program consists of only 5 additional lines of code in one function, but all the 75 non-empty functions in the resulting binary file are changed (see Sect. 6). To find semantic differences between such binaries, we must match semantically identical basic blocks that are nevertheless syntactically different and located differently, and similarly match semantically identical functions. In this paper, we use “match” and “matched pair” to denote a pair of basic blocks (or a pair of functions), one from each binary file, and use “matching” to refer to a set of matched pairs of basic blocks (or functions) in which any basic block (respectively, function) from the first file is matched to at most one from the second file, and vice versa.

We propose a novel technique, called BinHunt, to find the semantic differences between two binary files by analyzing the control flow of the program. The control flow reflects the functionality and seldom changes because of, e.g., different register allocations or basic block re-ordering, making it an attractive feature for finding the semantic differences. BinHunt first constructs a control flow graph (CFG) for each function and a callgraph (CG) for the entire binary file. After that, a customized graph isomorphism algorithm is used to find the best (partial) matchings between functions and between basic blocks. Our graph isomorphism algorithm is more accurate than previous techniques used in binary difference analysis, because its backtracking will replace erroneous matches by better ones, whereas previous approaches are greedy and erroneous matches will propagate through the isomorphism process. The output of this algorithm is a (partial) matching between functions in the two binary files, and a (partial) matching between basic blocks in two matched functions. It also outputs a *matching strength* for each match of functions or basic blocks, which tells how similar the two functions or basic blocks are. The matchings together with the matching strengths tell us where the semantic differences are.

A component of BinHunt is a method to accurately compare two basic blocks using symbolic execution and theorem proving. This novel technique helps determine if two basic blocks are functionally equivalent. It provides a guarantee that if two basic blocks are found to be different by BinHunt, then they must not be functionally equivalent. To the best of our knowledge, this is the first paper introducing such a technique for binary difference analysis. Being able to accurately compare two basic blocks not only improves the accuracy of the graph isomorphism computation, but helps in finding its solution faster.

We have implemented a system based on BinHunt and report on three case studies where we have used it to locate the semantic differences between binaries, which in these cases correspond to vulnerabilities in one version of a binary that was patched in the subsequent version. Note that when used to find software vulnerabilities, BinHunt is not meant to replace existing tools to generate exploits from syntactic binary differences [2], but instead to augment such tools. For example, in one of our case studies, all of the 75 non-empty functions in the binary change syntactically as the result of a very small change to one of the functions in the source code. Rather than applying an exploit-generation tool of Brumley et al. [2] to each of such a large number of syntactic differences to find the one that can be exploited, BinHunt can help identify the one semantic difference so that subsequent analysis can focus on that.

2 Problem Definition and Overview of Our Approach

Given are two binary files, which are usually two versions of the same program, possibly compiled with optimizations for increased performance. We assume that source code of these binary files is not available. We also assume that function names extracted from these binary files are unreliable for the purpose of binary difference analysis, since they can be changed easily.[1]

The outputs of the system are a (partial) matching between functions from the two binary files, a (partial) matching between basic blocks from two matched functions, and a *matching strength* for each pair of matched functions or basic blocks. The matching strength tells how similar the matched functions or basic blocks are. Unmatched functions and unmatched basic blocks, as well as matched functions and matched basic blocks with low matching strengths, constitute the semantic differences found between the two binary files.

The difference between two functions or between two basic blocks could be syntactic and semantic. Syntactic differences refer to differences in the instructions, whereas semantic differences refer to differences in functionality (i.e., input-output behavior). It is possible that a syntactic difference is not semantic.

[1] This is especially important when BinHunt is used to analyze malware, where attackers intentionally make static analysis of the software difficult.

As mentioned in Sect. 1, basic blocks could use different register allocations to perform the same task, which means that the two basic blocks with different instructions may have the same functionality. Similarly, basic block re-ordering may make two functions look very different, but the two functions could provide exactly the same functionality. Here we are concerned with finding semantic differences, and so the output matchings and matching strengths from BinHunt should eliminate functionally identical functions and identical blocks from consideration, by matching them to one another with high matching strengths.

BinHunt uses a graph isomorphism technique, applied to the control flow and call graphs of the two binary files, to find the partial matchings between functions and between basic blocks. In order to compute isomorphisms on graphs, BinHunt first needs to find the control flow of the programs. This is achieved by first disassembling the binary files and locating the code segments. The x86 instructions are then converted into an intermediate representation for constructing the control flow graphs and callgraphs. The graph isomorphism computation in BinHunt uses a novel technique to compare the functionality of basic blocks. This is achieved by symbolically executing the two basic blocks and using a theorem prover to test whether the effects of the basic blocks are always the same. With this novel technique for comparing the basic blocks, BinHunt is able to filter out most syntactic differences that do not correspond to semantic changes in the binary files.

3 System Architecture

Figure 1 shows the overall system architecture. The binary files are first passed to a front-end disassembler, which outputs a sequence of x86 instructions. Next, the x86 instructions are converted into an intermediate representation (IR) for easier and more accurate analysis. The IR is then used for control flow analysis, where the output is a set of control flow graphs (CFGs), one for each function, and a callgraph (CG) for the binary. In the last step, the CFGs and CG of the two binary files are passed to our graph isomorphism engine to find a matching between functions, a matching between basic blocks in matched functions, and matching strengths for each pair of matched functions and basic blocks.

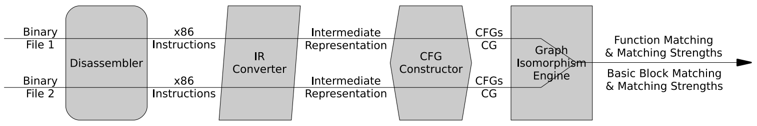


Fig. 1. Overall architecture of BinHunt

3.1 Disassembler

The disassembler parses each binary file and locates the code segments, which are disassembled into a sequence of x86 instructions. We implement a plug-in to

IDA Pro [6], a commercial disassembler, to do this, though other disassemblers could be used as well.

3.2 Intermediate Representation

The x86 computer family has a CISC (Complex Instruction Set Computer) instruction set, and is widely regarded as a very complicated set of instructions. For example, it consists of about 300 instructions; instructions are of variable length; and arithmetic instructions may set status bits, making them have side effects. It has undergone numerous changes over time, most of which were to add new functionality while maintaining backward compatibility. The complex nature of the x86 instruction set makes its static analysis difficult. In fact, some research even takes advantage of this complexity to obfuscate binary executables to improve their resistance to static analysis [10]. Because of this, some researchers choose to perform the analysis on an intermediate representation (IR), instead (e.g., [1]). In this project, we will take a similar approach, converting the x86 instructions into IR first, and then analyzing the IR.

The IR language we use is far simpler. It consists of roughly a dozen different statements, which are type-checked and free of side effects. In some cases, this simplicity does result in a loss of precision, and so our subsequent analysis might conclude that two basic blocks are functionally identical when they are not. However, we have not found examples where this loss of precision hides a semantic change in which we are interested. There are two major benefits of this simplicity. First, it makes the basic block comparison easier and more reliable. It is easier because our symbolic execution and theorem proving are applied on a much simpler set of instructions (see Sect. 4). It is also more reliable because the instruction simplification reduces the language variation in performing the same functionality. For example, two functionally equivalent basic blocks that are syntactically very different in their x86 instructions may look quite similar in their IR, because of the instruction simplification. Second, it makes our control flow analysis much simpler and more scalable (see Sect. 3.3).

3.3 Constructing Control Flow Graphs and Callgraphs

There are many ways of analyzing the IR of two binary files to find their differences. The IR of a program is just a sequence of instructions. Many traditional sequence comparison techniques can be used to find the differences between the two sequences, e.g., dynamic programming [14,12]. However, these techniques are not suitable in this paper, because our objective is not to find all syntactic differences in the instruction level, but to find semantic ones that correspond to changes in the program functionality.

What makes this problem unique is that there could be many changes in the instructions that do not correspond to changes in functionality, as described in Sect. 1. As a result, we propose analyzing the control flow of the programs to find the differences. The control flow of a program is much more resistant to "superficial" changes like different register allocations and basic block re-ordering, and therefore is a more attractive feature for finding semantic differences.

We represent the control flow of a program by a set of control flow graphs and a callgraph. A control flow graph (CFG) consists of a set of nodes each representing a basic block and a set of directed edges representing the control flow among the basic blocks. We construct a CFG for each function found in the binary file. We also construct a callgraph (CG) for each binary file, with the set of nodes corresponding to the functions in the file and the set of directed edges representing calls among the functions.

Note that although we base our binary difference analysis on the control flow of the program, the CFGs and the CG may not contain all necessary information for identifying the differences. For example, two programs may have isomorphic CFGs and CGs, but the nodes in them may be functionally different. Therefore, we still need to compare two corresponding basic blocks to find out if they provide the same functionality. Basic block comparison also helps in graph isomorphism (see Sect. 3.4), and we will detail our novel technique for basic block comparison in Sect. 4.

3.4 Comparing the CGs and the CFGs

A critical part of BinHunt is to compare the CFGs and CGs to find the matchings between corresponding functions and basic blocks. We do this by introducing a new graph isomorphism algorithm based on the backtracking technique.

Suppose we have obtained the CFGs of two functions by disassembling the code segments, converting x86 instructions into IR and analyzing the control flow. The next step is to compare the two CFGs to find nodes that match to each other. This can be conceptualized as the maximum common subgraph isomorphism problem.[2] Intuitively, the maximum common subgraph isomorphism problem is to find the largest common subgraph of two given graphs. Once the maximum common subgraph is found, nodes in the subgraph will correspond to matched basic blocks, and nodes outside of the subgraph will correspond to unmatched ones. Similarly, when given two CGs, we try to find the maximum common subgraph in which the nodes correspond to matched functions, and nodes outside of the subgraph correspond to unmatched functions.

We will detail our algorithm for finding the maximum common subgraph and how to interpret the output of the algorithm in Sect. 5. Note that two matched functions may not be exactly the same. There could be differences within the two functions and therefore we need to find the maximum common subgraph for the two matched functions as well in order to identify the binary differences. Since maximum common subgraph isomorphism is NP-complete, we need to propose an efficient algorithm which works practically for real problems. The efficiency of such an algorithm highly depends on the sequence in which nodes are examined for possible matches. In order to try the most likely matches first, we need to be able to tell whether the basic blocks represented by the nodes are similar. To do

[2] More precisely, we solve the maximum common *induced* subgraph isomorphism problem, which is defined in Sect. 5.1. However, we typically refer to this as simply the "subgraph isomorphism" problem or a similar variant, for simplicity.

this, we propose a novel technique using symbolic execution of the basic blocks and theorem proving. This technique is detailed in Sect. 4.

4 Basic Block Comparison

Basic block comparison is important for two reasons. First, it improves the efficiency of graph isomorphism. As mentioned in Sect. 3.4, the algorithm has better efficiency if the best matches are tried first, which is possible only if there is a way to measure the similarity of two basic blocks accurately. Second, basic block comparison helps identify any semantic differences between matched functions. As noted in Sect. 3.4, two matched functions or basic blocks may not be the same. Accurate basic block comparison can help identify the semantic differences with low false-positive and false-negative rates.

4.1 Symbolic Execution and Theorem Proving

Symbolic execution [7] is a well-known program analysis technique to represent values of program variables with symbolic values instead of concrete (initialized) data and to manipulate expressions involving symbolic values. For each basic block, we first find all the input and output registers and variables. We then use symbolic execution to represent the final values of the output registers and variables. That is, the output values computed by the basic blocks are expressed using the program input symbols. This process is fast since we are dealing with basic blocks, in which instructions are executed sequentially.

To test if two basic blocks are functionally equivalent, we apply a theorem prover to test if the output registers and variables of the basic blocks are the same. The theorem prover we employ is STP [5]. STP is a decision procedure for the satisfiability of quantifier-free formulas in the theory of bit-vectors and arrays that has been optimized for large problems encountered in software analysis applications. We pick the symbolic representation of one register/variable from each basic block and use STP to test if they are equivalent, assuming that the inputs to the basic blocks share the same values.

Symbolic execution with theorem proving helps us determine whether two registers or variables contain the same value after the executions of two basic blocks. However, in general we do not know which registers or variables to pick for testing because the two basic blocks could use different registers or variables to provide the same functionality. We try all pair-wise comparisons and check if there exists a permutation of the registers and variables between the two basic block such that all matched registers and variables contain the same value. If such a permutation exists, we conclude that the two basic blocks are functionally equivalent. With this, our technique ensures that if two basic blocks are found to be different by our technique of symbolic execution and theorem proving, then they must not be functionally equivalent.

Note that this property holds even if the two binary files are compiled using different compilers or compiler options. However, it only holds for basic block comparison and not for function comparison. That is, even if our symbolic execution and theorem proving show that the basic blocks in two functions are different, the two functions may, in fact, be functionally equivalent.

We could perform the same analysis for functions, i.e., we could use symbolic execution to represent outputs of two functions and test if these outputs are equivalent. However, performance becomes an issue as both symbolic execution and theorem proving take a long time to process functions of even moderate size.

4.2 Matching Strength

We define *matching strength* of two basic blocks to denote how similar they are in their functionality. Matching strength is a function of the two basic blocks only, and does not depend on the context in which they execute. Matching strength of two basic blocks that are deemed functionally equivalent (Sect. 4.1) is assigned 1.0 if they use the same registers or 0.9 if they use different ones. If two basic blocks are deemed not functionally equivalent, smaller matching strengths are assigned. The reason why we assign a matching strength of 0.9 for basic blocks using different registers is that our technique evaluates the basic blocks independently. Even if the two basic blocks are functionally equivalent, they may be used in different contexts. This is slightly more likely when the basic blocks use different registers, and therefore we assign a slightly smaller matching strength.

5 Maximum Common Induced Subgraph Isomorphism

Graphs are sets of nodes with edges connecting pairs of nodes. Similarity measurement between graphs can be achieved by graph matching, which is a procedure to identify common subgraphs. The measure of similarity is then given by the size of the maximum common subgraph. This technique has its application to many problems.

Graph matching is known to be a computationally expensive procedure. A number of graph-matching algorithms, both optimal and approximate, have been proposed over the last three decades [11]. Most optimal algorithms for common subgraph isomorphism are based on maximal clique detection in the association graph [9]. However, it is widely accepted that the problem of exact subgraph isomorphism, which is a special case of common subgraph isomorphism with the requirement that the resulting common subgraph coincides with one of the input graphs, is much more efficiently solved by the backtracking algorithm [8,13].

In our problem of binary difference analysis, especially in applications in which the two binary files under analysis are two versions of the same program, the maximum common subgraph is very likely the same or very close to one of the two input graphs. Therefore, we choose an algorithm based on the backtracking technique [8]. Below, we first define the common subgraph problem we are trying to solve. We then describe the basic idea of the backtracking algorithm, and our customizations to it to make it efficient and suitable for binary difference analysis.

5.1 Definitions

Given a graph $G = [V, E]$, graph $H = [W, F]$ is an induced subgraph of G if and only if $W \subseteq V$ and $F = E \cap (W \times W)$. Given two graphs G and H, we define the *maximum common induced subgraph isomorphism* problem as finding the largest induced subgraph of G that is isomorphic to an induced subgraph of H. We call this largest induced subgraph the *maximum common induced subgraph* of G and H. Here "largest" means that the subgraph is largest according to some *subgraph measurement*, which is not necessarily as simple as counting the number of nodes in the subgraph. We define this subgraph measurement in Sect. 5.3.

5.2 Backtracking Algorithm

The backtracking algorithm offers a simple solution to the maximum common induced subgraph isomorphism problem. The algorithm essentially enumerates all possible matches of the nodes from the two input graphs. A property of this algorithm is that an erroneous match added to the result will be replaced by a better match subsequently.

```
Isomorphism ($D, M$)
 1: if Extendable($D, M$) then
 2:    $v \leftarrow$ PickAny($D$)
 3:    $Z \leftarrow$ GetPossibleMatching($v, D$)
 4:    for all $w \in Z$ do
 5:       $M' \leftarrow M + [v, w]$
 6:       $D' \leftarrow$ Refine($v, w, D$)
 7:       Isomorphism($D', M'$)
 8:    end for
 9:    $D' \leftarrow$ Refine($v$, null, $D$)
10:    Isomorphism($D', M$)
11: end if
```

Fig. 2. Isomorphism function

Figure 2 shows the pseudo-code of a recursive version of the backtracking algorithm. If $G = (V, E)$ and $H = (W, F)$ are the input graphs, then D contains all possible pairs of nodes that might still be matched (initially $V \times W$), and M contains matched node pairs (initially empty). We assume that G and H are global variables in Figure 2; all other variables are local and are passed by value. On each entry to `Isomorphism()`, M records the matched node pairs of the partial common induced subgraph found. It first checks whether the solution is extendable. If it is extendable (Line 1), it picks an unmatched node v (Line 2) and assigns Z all possible matches for v (Line 3). For each node w in Z (Line 4), the algorithm extends the solution by adding the match $[v, w]$ to M (Line 5), refining D (Line 6), and recursively calling `Isomorphism()` (Line 7). In the end, the search is complemented by exploring extensions of M that do not include the chosen node v (Line 9 and 10). This last step is necessary because a subgraph not containing v (considered in the last step) may be larger than any subgraphs containing v (considered in earlier steps). With this, `Isomorphism()` completes trying all possible matches of the nodes.

5.3 Customizations to the Backtracking Algorithm

Matching strength and subgraph measurement. In traditional backtracking algorithms [13,8], nodes are labeled (colored) and a possible match is one that consists of nodes with the same label (color). We do not introduce such labels for nodes in our CFGs and CGs, because any nodes in a graph may be matched with any nodes in the other graph in our problem. Instead, we utilize matching strength to guide us which two nodes are more likely to be matched.

The matching strength for two basic blocks was detailed in Sect. 4.2. This is the matching strength used when computing the maximum common induced subgraph of the CFGs of two functions. We define the matching strength of two functions—which is used when computing the maximum common induced subgraph of the CGs of two binaries—to be 1.0 if the instructions (x86 or IR) of the two functions are the same. Otherwise, their matching strength is the *subgraph measurement* divided by the number of nodes in the CFG that has fewer nodes, where *subgraph measurement* is defined as the summation of matching strengths of matched nodes (basic blocks). Subgraph measurement is used not only in defining the matching strength of two functions, but in the definition of maximum common induced subgraph (see Sect. 5.1).

Customizations to improve efficiency. As the backtracking algorithm enumerates all possible matches, it could be very inefficient for large graphs. The functions `Extendable()`, `PickAny()` and `Refine()` in Figure 2 are important in making the algorithm efficient for practical problems.

Function `Extendable()` first makes a prediction on the maximum common induced subgraph assuming all unmatched nodes can be matched perfectly, which is the best possible output given the current matching. If this best possible output is not better than the best subgraph found previously, then further enumeration is not necessary and `Extendable()` simply returns false.

`Extendable()` can improve the efficiency only if good matches are tried first. The earlier the good matches are tried, the more times when `Extendable()` returns false, and therefore the more efficient the algorithm is. So we want to make function `PickAny()` return the best matching candidate first. In BinHunt, `PickAny()` returns the node that has the largest matching strength with nodes in the other graph. `PickAny()` also considers the connectivity (number of predecessors and successors) of the node; i.e., nodes with larger connectivity will be returned if there are multiple nodes with the same maximum matching strength.

`Refine()` updates the set of possible future matches by removing 1) $[v, w]$ (the new match); and 2) other matches that would not conform to the definition of common induced subgraph were they added to M in the future. For example, assume that v has a predecessor v' in G ($(v', v) \in E$), and w' is not a predecessor of w in H ($(w', w) \notin F$). $[v', w']$ has to be removed from D because if

$[v', w']$ is added to M, then the subgraph of G will contain edge (v', v) while the subgraph of H will not contain edge (w', w), making the two subgraphs not isomorphic.

Timeout and output of the backtracking algorithm. With the help of the three functions `Extendable()`, `PickAny()` and `Refine()`, the backtracking algorithm is able to try the best match early and therefore becomes more efficient by quickly terminating the processing of other possible matches that will not result in large subgraphs. However, in some cases where the CFGs are very big, the algorithm may still take too much time to converge.

To have a good balance between efficiency and accuracy, we introduce a timeout on some invocations to `Isomorphism()`. Specifically, when BinHunt tries to find the matching strength for every pair of functions from the two binary files for the input to CG isomorphism, timeouts are enabled. If a timeout is reached, we simply assign a default value to the matching strength. In CG isomorphism, however, the timeout is disabled. Lastly, after the function matching is found, the maximum common induced subgraph is recalculated (with timeout disabled) for matched function pairs that resulted in a timeout in the first step.

The output of BinHunt consists of the (partial) matching between functions from the two binary files, the (partial) matching between basic blocks for matched functions, and the matching strengths for the matched functions and basic blocks. Note that semantic differences correspond to unmatched functions and basic blocks, as well as matched ones with low matching strengths.

6 Case Studies

We implemented a system for binary difference analysis with the above components and techniques. In this section, we will show the results of using this system in three case studies to discover vulnerabilities in `gzip`, `tar` and `ASP.NET`. We specifically choose these three cases to show how BinHunt performs in complex cases where a small change in one function in the source code leads to substantial syntactic changes in many other functions (the case of `gzip`), when the semantic changes result in change of control flow in the program (the case of `tar`), and when only the binary files are available (the case of `ASP.NET`). For the first two case studies, we obtained the source code of the patched and unpatched versions and compiled them independently to obtain the binary executables for analysis. Once the binary executables were obtained, we made no further use of the source code. For `ASP.NET`, we downloaded the patched and unpatched binaries directly from the software vendor.

Note that in all of our case studies, the only differences between the two versions correspond to patching vulnerabilities. In other cases, the differences may correspond to new features added to the program, which would complicate the discovery of vulnerabilities.

6.1 Buffer Overflow in gzip

A release of gzip has a vulnerability that causes a buffer overflow with a file name of 1028 bytes or greater. (See http://www.securityfocus.com/bid/3712 for more details.) This overflow could overwrite stack variables and return addresses, possibly resulting in arbitrary code execution. A patch exists for fixing the problem, and is shown in Fig. 3.

```
 #ifdef NO_MULTIPLE_DOTS
     char *dot; /* pointer to ifname extension, or NULL */
 #endif
+    int max_suffix_len = (z_len > 3 ? z_len : 3);
+    /* Leave enough room in ifname or ofname for suffix: */
+    if (strlen(iname) >= sizeof(ifname) - max_suffix_len) {
+        strncpy(ifname, iname, sizeof(ifname) - 1);
+   /* last byte of ifname is already zero and never overwritten */
+        error("file name too long");
+    }
     strcpy(ifname, iname);
     /* If input file exists, return OK. */
```

Fig. 3. The patch for gzip

Without going into too much detail, we can see that the patch checks the length of the variable iname and outputs an error message when it is too long. The patch adds a few statements in a small function get_istat(), which corresponds to only 7 additional lines (including two comment lines). The original source of gzip.c contains 1,744 lines, which means that the change accounts for roughly a 0.4% change in the source code (when not considering other dependencies). Although it is a very small change in the source code, we find that none of the 75 non-empty functions in the patched binary (which contains more than 8,500 instructions) is syntactically the same as any functions in the unpatched binary. Further analysis shows that this is mainly due to basic block re-ordering.

Although all non-empty functions are changed syntactically, BinHunt was able to find the correct matching for all non-empty functions. Figure 4 shows the number of non-empty functions that had matching strengths less or equal to the value on the x-axis. Despite the fact that all non-empty functions contain syntactic changes, BinHunt managed to find more than 10 function matches that have matching strengths of 1.0, which means that these matched functions contain basic blocks that are functionally equivalent and that use the same register allocation. There is also a large number of function matchings that have matching strengths between 0.9 and 1.0, which means that these matched functions contain basic blocks that are functionally equivalent, although some of these basic blocks use different register allocations. (See definitions of matching strength in Sect. 4.2 and Sect. 5.3.)

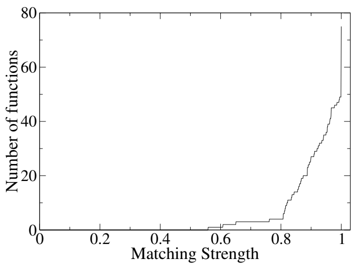


Fig. 4. Matching strengths of functions in gzip (CDF)

Among the matched functions with a matching strength less than 0.8, there was only one pair that differed substantially (by 26) in the number of unmatched basic blocks, which was function `treat_file()`. The rest of the matched functions had zero or a very small number of unmatched basic blocks. However, `treat_file()` is not the function `get_istat()` where changes were made in the source code, but rather its parent function, because function inlining was applied during compilation in both patched and unpatched versions. The parent function is a very large function with more than 900 basic blocks. BinHunt was able to find common subgraphs of a size almost the same as the unpatched function (all basic blocks were matched correctly except the two nodes between which new basic blocks are added), and therefore identified a few additional basic blocks that exist only in the CFG of the patched binary (see Fig. 5 for the additional basic blocks in the dotted rectangle).

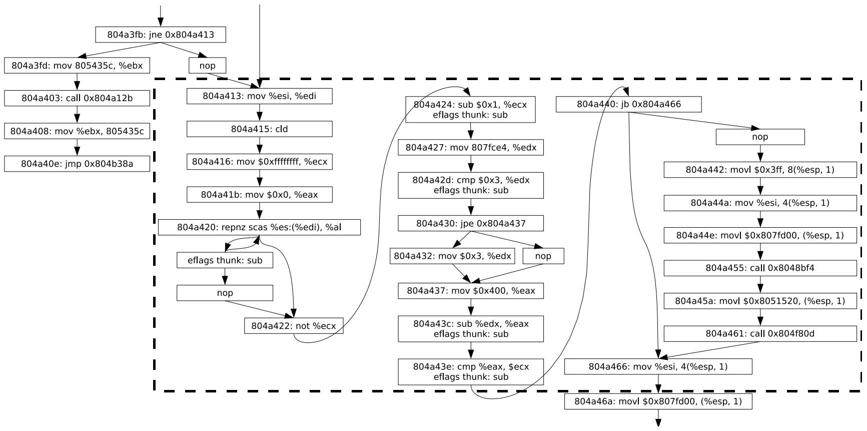


Fig. 5. Difference found in the parent function of `get_istat()`

```
mov %esi, %edi
cld
mov $0xfffffff, %ecx
mov $0x0, %eax
repnz scas %es:(%edi), %al
```

Fig. 6. Assembly code for differences found in `gzip`

Looking at the first few basic blocks within the dotted rectangle in Fig. 5, we found the assembly code as shown in Fig. 6. The `repnz` instruction repeats doing something as long as a byte is non-zero, which, in most cases, is used to find the terminating byte in a string. In this case study, it is used for the same purpose to find the length of the string at `edi`. By tracing the register `edi` we easily found that the string is actually the input parameter representing the file name.

In this case study, BinHunt identified a few basic blocks that correspond to changes in the source code of `gzip`. It took about an hour to finish the analysis on a desktop computer with a 2.1GHz CPU. The reason why it takes relatively long is because all functions in the patched version are syntactically different from functions in the unpatched version. In this case, we configured BinHunt to perform graph isomorphism on all permutations of the 75 functions in each binaries. This also means that the analysis time would not increase substantially if there were more semantic differences between the two binaries.

6.2 "Dot_dot" Vulnerability in `tar`

The patch for the buffer overflow vulnerability in `gzip` inserts additional instructions into a function, which result in a few additional basic blocks in a CFG. However, these additional basic blocks do not alter the control flow of the original program. BinHunt is very powerful in identifying this type of change in source code because our binary difference analysis is based on analysis of the control flow of the program. Although many patches share the same characteristics of the one for `gzip`, there are patches which change the control flow of the program. In this subsection, we describe an application of BinHunt to a program in which the patch changes the control flow of the original program.

```
bool contains_dot_dot (char const *name) {
  char const *p = name +
    FILE_SYSTEM_PREFIX_LEN (name);
  for (;; p++) {
    if (p[0]=='.' && p[1]=='.' &&
      (ISSLASH(p[2]) || !p[2]))
      return 1;
    do { if (! *p++) return 0; }
      while (! ISSLASH (*p));
  }
}
```

Unpatched

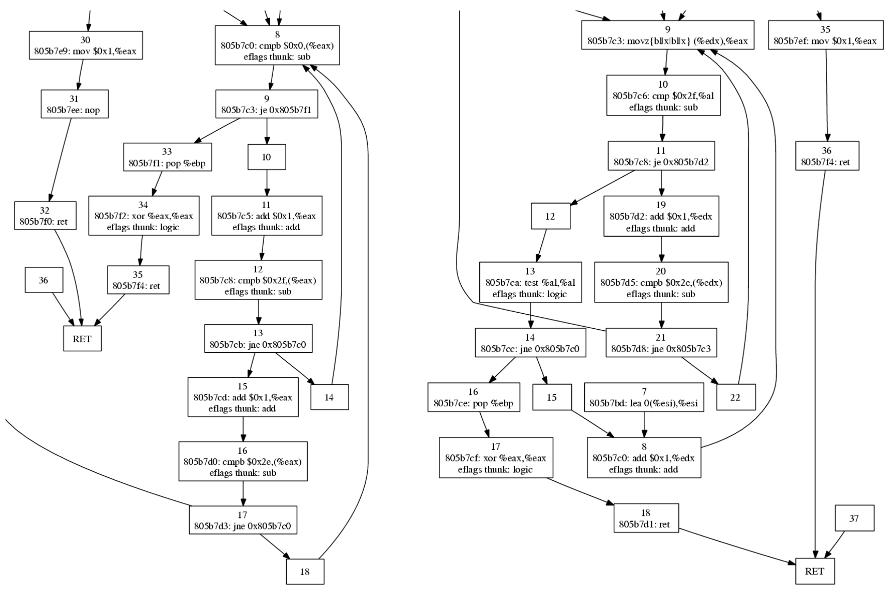
```
bool contains_dot_dot (char const *name) {
  char const *p = name +
    FILE_SYSTEM_PREFIX_LEN (name);
  for (;; p++) {
    if (p[0]=='.' && p[1]=='.' &&
      (ISSLASH(p[2]) || !p[2]))
      return 1;
    while (! ISSLASH (*p))
      { if (! *p++) return 0; }
  }
}
```

Patched

Fig. 7. The unpatched and patched functions in `tar`

A version of the program `tar` has an input validation error called the "dot_dot" function vulnerability. (See `http://www.securityfocus.com/bid/25417/info` for more information.) Attackers may exploit the vulnerability to overwrite files on the computer. Figure 7 shows the unpatched and patched versions of the function in which the vulnerability is found. The difference in the source code is that the unpatched function uses a `do-while` loop, whereas the patched function uses a `while` loop. This patch changes the control flow of the program. Again, BinHunt was able to find the correct matching between all non-empty functions from the two binary files. In this case, more than 95% of the matches had a matching strength of 1.0. This is due to the fact that function `contains_dot_dot()` is located towards the end of the binary file, and changes in it do not result in much basic block re-ordering. Out of the 470 non-empty functions, the match for function `contains_dot_dot()` had the smallest matching strength of 0.571622.

The graph isomorphism calculation on the CFGs for function `contains_dot_dot()` found a matching for about two-thirds of the basic blocks (the unpatched version has 36 basic blocks and the patched version has



30 805b7e9: mov $0x1,%eax
31 805b7ee: nop
32 805b7f0: ret
36
RET
8 805b7c0: cmpb $0x0,(%eax) eflags thunk: sub
9 805b7c3: je 0x805b7f1
33 805b7f1: pop %ebp
10
34 805b7f2: xor %eax,%eax eflags thunk: logic
35 805b7f4: ret
11 805b7c5: add $0x1,%eax eflags thunk: add
12 805b7c8: cmpb $0x2f,(%eax) eflags thunk: sub
13 805b7cb: jne 0x805b7c0
15 805b7cd: add $0x1,%eax eflags thunk: add
14
16 805b7d0: cmpb $0x2e,(%eax) eflags thunk: sub
17 805b7d3: jne 0x805b7c0
18
9 805b7c3: movz{bl|x|bl|x} (%edx),%eax
35 805b7ef: mov $0x1,%eax
10 805b7c6: cmp $0x2f,%al eflags thunk: sub
11 805b7c8: je 0x805b7d2
36 805b7f4: ret
12
19 805b7d2: add $0x1,%edx eflags thunk: add
13 805b7ca: test %al,%al eflags thunk: logic
20 805b7d5: cmpb $0x2e,(%edx) eflags thunk: sub
14 805b7cc: jne 0x805b7c0
21 805b7d8: jne 0x805b7c3
16 805b7ce: pop %ebp
15
7 805b7bd: lea 0(%esi),%esi
22
17 805b7cf: xor %eax,%eax eflags thunk: logic
8 805b7c0: add $0x1,%edx eflags thunk: add
18 805b7d1: ret
37
RET
(a) Unpatched
(b) Patched

Fig. 8. Part of the CFGs of function `contains_dot_dot()`

37 basic blocks). Figure 9 shows the number of basic blocks that were matched with a matching strength less than the value on the x-axis. The analysis of the unmatched basic blocks to identify the vulnerability required a little more effort because of the changes in control flow. Figure 8 shows part of the CFGs of the unpatched and patched function `contains_dot_dot()`.

As can be seen from Fig. 8, a loop exists in both CFGs, i.e., the path between basic block number 8 and 18 in the unpatched function and the path between basic block number 9 and number 8 in the patched function. There is a special basic block that does the comparison of a byte with the character "/" (ASCII `0x2f`); this is basic block 12 in the unpatched function and basic block 10 in the patched version. It can been seen from the CFGs that the comparison is performed in the middle of the loop in the unpatched function, and at the beginning of the loop in the patched function. With this, the vulnerability is found.

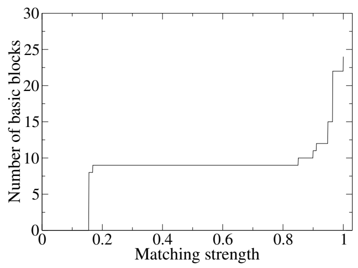


Fig. 9. Matching strengths of basic blocks in `contains_dot_dot()` (CDF)

In this case study, there are more than 41,000 instructions in each of the two binaries. It took about 30 minutes for BinHunt to finish the analysis.

One of the reasons why it took only 30 minutes is that some of the functions in the two binaries are exactly the same, and so BinHunt did not need to perform the graph isomorphism for these functions.

Another interesting thing to note is about different register allocation. Figure 10 shows the first few basic blocks of function `contains_dot_dot()` for both the unpatched and patched versions. We can see from basic block number 2 that `eax` is used in the unpatched function, while `edx` is used in the patched function for the same purpose. This is an example of syntactic differences that BinHunt managed to skip when finding semantic differences.

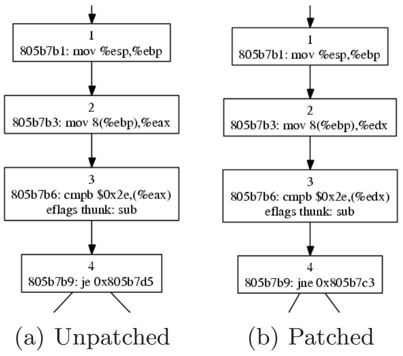


Fig. 10. Different register allocation for function `contains_dot_dot()`

6.3 Application Folder Information Disclosure in `ASP.NET`

The last case study we did was on Microsoft .NET framework 2.0 (`ASP.NET`). Unlike the previous two cases in which we compiled the source to obtain the binary executables independently, in this case study we downloaded the binary files directly from the software vendor. `ASP.NET` in many versions of Microsoft Windows allows remote attackers to bypass access restrictions via unspecified "URL paths" that can access Application Folder objects "explicitly by name" (CVE-2006-1300). This vulnerability occurs because `ASP.NET` only checks for slash ("/") and does not consider `%5c` (the ASCII code for "\") when checking for accessibility.

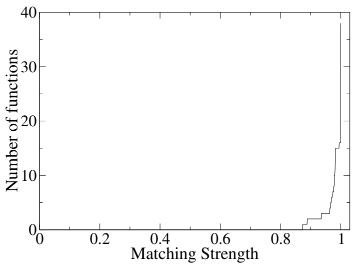


Fig. 11. Matching strengths of functions in `ASP.NET` (CDF)

BinHunt found that there are 38 non-empty functions in the unpatched versions of the binary files and 39 non-empty functions in the patched version, and found the correct matching for 38 functions. Figure 11 shows the number of functions that were matched with a matching strength less than the value on the x-axis. We can see that all matched functions had a high matching strength.

In this case study, it was trivial to find the semantic difference as it corresponds to an unmatched function `FlipSlashes()`, which is called from function `HttpFilterProc()` to perform additional checks. BinHunt managed to locate the unmatched basic block in function `HttpFilterProc()` which corresponds to the call of `FlipSlashes()`. This case study shows that BinHunt works as expected on binary files downloaded directly from the software vendor.

7 Related Work

The structural comparison tools BinDiff [4] (and its extension [3]) and BindView (`http://www.bindview.com/Services/Razor/Papers/2004/comparing_binaries.cfm`) are most related to our work. These tools construct a maximal subgraph isomorphism between the sets of functions in two versions of the same executable file. There are two major distinctions between these systems and BinHunt.

First, BinHunt contributes a more thorough technique for identifying the maximum common subgraph isomorphism. BinDiff and BindView use a greedy method to extend a matching, and thus an erroneous match will propagate, leading to a failure to find the maximum subgraph isomorphism. In comparison, BinHunt uses a backtracking technique to find the maximum isomorphic subgraphs (see Sect. 5). While in general this would be exceedingly expensive, we develop optimizations to make it practical. Inaccurate matches added to the result will be replaced by better ones subsequently in the backtracking process.

Second, BinHunt uses a novel technique for basic block comparison using symbolic execution and theorem proving (see Sect. 4). This method can determine if two basic blocks are functionally equivalent, which overcomes the difficulty encountered when, e.g., basic blocks use different register allocations. In contrast, BinDiff uses heuristics to test if two graphs or basic blocks are similar. For example, BinDiff compares two graphs by calculating the number of basic blocks, edges and callers. BindView matches basic blocks based on instructions present in them. Due to the reliance on comparing actual instructions, a significant number of locations are falsely identified as changes [3].

There are also binary difference analysis tools to produce and apply patches (bsdiff, bspatch, xdelta, jdiff, jpatch, etc.). They capture all syntactic differences between binaries; as described previously, such differences may not correspond to semantic differences, and so they do not suffice for the goals of this paper.

8 Conclusion and Limitations

In this paper, we define the problem of finding semantic differences in binary executables, and introduce a novel technique BinHunt based on control flow analysis. When compared with previous techniques, BinHunt uses a more thorough graph isomorphism technique for identifying the maximum common induced subgraph isomorphism. Unlike previous techniques, BinHunt does not rely on many heuristics when finding the maximum common subgraph. BinHunt also makes use of a novel technique to compare the functionality of two basic blocks using symbolic execution and theorem proving. In case studies on different versions of three common programs, we showed that BinHunt is able to find the semantic differences with high accuracy.

A limitation of BinHunt is that its analysis efficiency drops when the number of semantic differences between binary files increases. This is due to the graph isomorphism technique that BinHunt uses. The backtracking algorithm works the best when the two graphs are similar to each other. In applications where the differences between the two binary files are large, a different graph isomorphism technique should be used. BinHunt does not work on packed code, either. We leave these topics for future work.

References

1. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: Codesurfer/x86 - a platform for analyzing x86 executables. In: Proceedings of the Conference on Compiler Construction (2005)
2. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: techniques and implications. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (May 2008) (to appear)
3. Dullien, T., Rolles, R.: Graph-based comparison of executable objects. In: Proceedings of SSTIC 2005 (2005)
4. Flake, H.: Structural comparison of executable objects. In: Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment 2004 (2004)
5. Ganesh, V., Dill, D.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590. Springer, Heidelberg (2007)
6. DataRescue Inc. IDA Pro, `http://www.datarescue.com/idabase/`
7. King, J.: Symbolic execution and program testing. Communications of the ACM 19(7) (1976)
8. Krissinel, E., Henrick, K.: Common subgraph isomorphism detection by backtracking search. Software — Practice and Experience 34 (2004)
9. Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcolo 9 (1972)
10. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 11th ACM Conference on Computer & Communication Security (CCS 2003) (2003)
11. Raymond, J., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. Journal of Computer-Aided Molecular Design 16 (2002)
12. Sankoff, D., Kruskal, J.B.: Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison. Addison-Wesley Pu. Co., Reading (1983)
13. Ullman, J.: An algorithm for subgraph isomorphism. Journal of the Association of Computers and Machines 23 (1976)
14. Vintsyuk, T.K.: Speech discrimination by dynamic programming. Cybernetics and Systems Analysis 4(1) (1968)
15. Wang, Z., Pierce, K., McFarling, S.: Bmat - a binary matching tool for stale profile propagation. J. Instruction-Level Parallelism 2 (2000)