SCV: Structure and Constant Value based Binary Diffing

Heewan Park  Seokwoo Choi  Sunae Seo  Taisook Han
Korea Advanced Institute of Science and Technology
335 Gwahangno Yuseong-gu, Daejeon 305-701, Republic of Korea
{hwpark, swchoi, saseo, han}@pllab.kaist.ac.kr

Abstract

Binary diffing is a method to find differences in similar binary executables such as two different versions of security patches. Diffing methods using flow information detect control flow changes very fast, but they cannot track constant value changes. We present a binary diffing tool named SCV which utilizes both structure and value information. SCV summarizes structure and constant value information from disassembled code, and matches the summaries to find differences. By analyzing a Microsoft Windows security patch KB938827, we showed that SCV found necessary differences caused by constant value changes which the state-of-the-art binary diffing tool BinDiff failed to find.

1. Introduction

In recent years, binary code diffing tools have been widely used to analyze differences between similar binaries. Binary diffing tools analyze program patches of operating systems, malware variants, the codes which allegedly infringed copyrights, etc. Analyzing security patches is especially useful when the whole or a part of the changes are not announced by patch provider.

When comparing two assembly codes obtained from binaries, we consider three aspects:

1. We need to analyze a structure shape of the assembly code. Simple linear comparison of two assembly codes does not provide the suitable change information. The assembly code consists of a series of instructions which can be divided into modules such as a function or a basic block. Without affecting the control flows, those modules can be placed in the different physical locations in the file. The structure information of the assembly code allows us to compare the corresponding modules regardless their physical locations.

2. We should not miss important instruction-level changes such as resized buffers and loop counts, but in the meanwhile, we hope to ignore changes from less-aggressive optimizations such as instruction reordering and register reallocation.

3. Finally, we need to quantify the match ratio when the complete match does not occur. This quantified match ratio allows the user to estimate the amount of the relative difference.

We explain those aspects using the example in Figure 1. The assembly codes in Figure 1 are obtained after disassembly of two different files of GDI32.dll binaries, which is related to the remote code execution vulnerability†. Although a quick code inspection may not find suitable matches between them, our tool SCV finds that the left code part is changed to the right code part. Note that simple linear comparison cannot discover this fact, because two codes have totally different sequence of instructions, different register names, different jump addresses, etc. In order to find a basic block match, SCV first computes control-flow graphs for each function, and performs structure comparison between basic blocks. The control-flow graphs for the codes in Figure 1 are shown in Figure 3.

A demerit of the top-down style structure match is that it does not find instruction-level changes. We want to trace the important instruction-level changes efficiently such as changes of buffer sizes and loop counts. In the meanwhile, we hope that our tool ignores less-aggressive optimizations such as instruction reordering and register reallocation. For this purpose, SCV find immediate values from the binaries and uses them for comparison. For example, in block B1.02 and B2.02 of Figure 1, there are different registers and different jump addresses used. SCV does not alarm those differences, but it reports two immediate

†This Microsoft security patch is KB938827.
values $7FFFFFC8h$ and $0FFFFFFDh$ are different. SCV reports that block $B_{1\_05}$ is changed to $B_{2\_05}$ by inserting one instruction. For the blocks $B_{1\_03}, B_{1\_04}, B_{2\_03}$, and $B_{2\_04}$, SCV informs that there is no complete match, but by inspecting the control-flow graph that SCV provides the user can complete the matches between them. Moreover, SCV computes the match rates for function structures and constant values. For the pair of functions in Figure 1, SCV concludes that the function structure match is 1.0 which means their structures are the same, and that the constant value match is 0.22 which means that there is large difference in constant values. This fact is shown in Figure 2 and we will explain this figure in Section 3 in detail.

2. SCV Binary Differing

2.1. Structure Comparison

For structure comparison, SCV uses graph-based comparison in [2, 3]. Binary executables are disassembled using IDAPro disassembler [1]. After distinguishing functions and basic blocks, we analyze function call relations for function match and control-flow graphs for basic block match. For a fast match, we collect function summaries as well as block summaries, and perform appropriate matches using the summaries. A function summary is a tuple of three integers $(i, j, k)$, where $i$ is the number of basic blocks in the function, $j$ is the number of control-flow edges between basic blocks and $k$ is the number of function calls occurring in the function. A basic block summary is defined similarly using the relative location of the block in the function. For space limitation, we omit details of the summaries and the structure match.

2.2. Constant Value Comparison

On comparing binaries, an important modification other than structure changes is to alter buffer sizes or loop counts. Modifications on those constants can influence the binary execution traces. In order to detect the changes on those constants, SCV collects immediate values from binaries and makes summaries of them for comparison. The constant summaries consist of the occurrences of the immediate values. For a function $f$, constant summary $\delta(f)$ is defined as follows:

$$\delta(f) = \{(v_1, n_1), (v_2, n_2), ..., (v_n, n_n)\},$$

where $v_i$ is an immediate value occurring in $f$, and $n_i$ is the number of occurrence of $v_i$ in $f$. For example, when a function $f$ uses constant $0x800$ two times and $0x100$ one time, $\delta(f) = \{(0x800, 2), (0x100, 1)\}.$

2.3. Match Rates

SCV provides two kinds of match rates: one is the match rates for function structures, and the other is for constants in functions. The match rates for function structures are computed by the cosine measure of the function summaries. Let $\alpha(f)$ be the function summary of a function $f$. Given two functions $f$ and $g$ with $\alpha(f) = (i_1, j_1, k_1)$ and $\alpha(g) = (i_2, j_2, k_2)$, their match rate is defined as follows:

$$\text{Match rate for function structures} = \frac{\alpha(f) \cdot \alpha(g)}{|\alpha(f)| \cdot |\alpha(g)|} = \frac{i_1 i_2 + j_1 j_2 + k_1 k_2}{\sqrt{i_1^2 + j_1^2 + k_1^2} \cdot \sqrt{i_2^2 + j_2^2 + k_2^2}}$$
The match rates for function structures are mathematically the cosine of two three-dimensional vectors. Note that the match rate 1.0 does not mean two functions are the same. Since the given match rates are defined by the function summary, two functions with the same function summaries but having different details do not have the match rate 1.0. For example, though the control-flow graphs in Figure 3 are different, those two functions have match rate 1.0 for function structures.

The match rates for constants complement the match for function structures. Since the match rates for function structures do not provide any information of constant values, we define the match rates for constants. Suppose that the constant summaries of two function $f$ and $g$ are given as follows:

$$
\delta(f) = \{(v_1, q_1), (v_2, q_2), ..., (v_m, q_m)\}
$$

$$
\delta(g) = \{(w_1, r_1), (w_2, r_2), ..., (w_n, r_n)\}.
$$

Then, we define the match rates for constants as follows:

$$
\text{Match rate for constants } = \frac{2|\delta(f) \otimes \delta(g)|}{|\delta(f)| + |\delta(g)|},
$$

where $|\delta(f)| = \sum_{i=1}^{m} q_i$, $|\delta(g)| = \sum_{i=1}^{n} r_i$, and

$$
\delta(f) \otimes \delta(g) = \{(v, \min(q, r))|(v, q) \in \delta(f) \land (v, r) \in \delta(g)\}.
$$

The match rates for constants are calculated using Dice similarity measure. The numerator is the number of identical constants in two functions, and the denominator is the number of all constants in two functions. For example, the match rate of constants in two functions is 0.22, which indicates that constant values in two functions are fairly different.

### 3. Implementation

The implementation of SCV is composed of three parts. The first part is a frontend that accepts binary executables and outputs SQLite DB files, which is implemented using IDAPro disassembler and IDA2SQLite plug-in. The second part is a binary code match engine. SCV computes structure data including the call-graph relation and the control-flow information, and performs the function match, the basic-block match, and the constant value match. The third part is the visualization routine that displays the match results on the screen. This part is implemented using GraphViz library [4].

For example, we present parts of SCV screen shot in Figure 2 and Figure 3, which is the comparison result of two different GDI32.dll files. This patch comes from Microsoft update MS07-0461. This vulnerability may allow remote code execution. Figure 2 shows the function match result that SCV displays, and Figure 3 shows the basic-block match in the form of control-flow graph. In Figure 2, the match rates for structures and constant values for all matched functions are presented. Moreover, the match results for basic blocks and assembly codes of them are all depicted on the screen, but we omit those parts for space limitation.

In Figure 2, the function `GetEvent` from two files have a structure match rate less than 1.0. SCV shows that 7 conditional statements are added for fear of the remote execution vulnerability. Moreover, from the constant match rates, we know that constants in 9 functions are modified. SCV visualizes exactly which functions, basic blocks, and immediate values have changed by coloring the corresponding assembly codes as well as control-flow graphs. In Figure 3, we show the screen shot that SCV depicted, which presents the difference in control-flow graphs for the 5th function of the function match table (Figure 2). The white pairs of basic blocks such as (B1.02, B2.02), (B1.06, B2.06), etc mean that SCV found a structure match for each pair of the blocks and that the instructions in the blocks

---

are the same. The light grey pairs of basic blocks such as (B1_01, B2_01), (B1_05, B2_05) and (B1_09, B2_09) present that SCV found a structure match between them but the instructions in the blocks are different. The dark grey pairs such as (B1_03, B2_03) and (B1_04, B2_04) mean that SCV could not find a structure match. The hexagon nodes indicate that SCV found constant value difference. The constant match rates for each function are found in the third column of the function match table in Figure 2. The existing commercial binary difference tool, BinDiff[6], cannot identify this changes.

4. Conclusion

In this paper we presented a novel binary diffing tool named SCV. Our tool summarizes both control structures and constant values of binary executables and matches functions and basic blocks with the summaries. By analyzing a Microsoft Windows security patch, we showed that SCV can find necessary differences caused by constant value changes which the state-of-the-art binary diffing tool BinDiff failed to find. However packed or obfuscated code [5] is difficult to be compared by our tool, because we use IDAPro disassembler as a front-end, who has limited power against these cases. Further research on the unpacker and deobfuscator would help us improve the current comparing technique, and hence it could be used to detect the malware-infected programs.

References