Analyzing Malware Putty using Function Alignment in the Binary

Arne Wichmann, Sandro Schulze, Sibylle Schupp
Technische Universität Hamburg-Harburg, Hamburg, Germany
{arne.wichmann, sandro.schulze, schupp}@tuhh.de

This paper shows a representation of executables and an alignment of functions in an executable to be used when reverse engineering embedded systems. These techniques are not limited to this application and can also be used when studying code variations, code clone-and-own scenarios, and when locating hotspots for software quality inspections.

1 Techniques

The proposed applications make use of one already presented technique, namely a scalable representation using scatter plots of the static control flow of executables [7] and an alignment technique based on Hirschberg [4], which we have not presented so far. This alignment technique produces an alignment of two function sequences using a similarity measure on the optionally normalized function lengths, which is based on a previous work that employed more metrics to produce the alignment [6].

The key assumption behind these techniques is that the sequence of functions in an executable is mostly stable and invariant against changes in compiler version, optimisation level, source-code version, and the selected feature set. For example, in a classical C executable the code from one source file is transformed into an object file and several of these, together with some libraries are linked into one executable, without any further changes to their order.

Scalable Representation The scatter plot representation [7] shows all static control flow references of the executable. The raw data used for this are all pairs of program counters and next possible program counters. This covers the intraprocedural control flow, as well as calls to other functions.

The presentation can use several different scales on the axes which allow it to be invariant to different kinds of distortions: It always plots pairs of source and target addresses of a control reference. A raw address representation hides the length of the functions in the executable. These function lengths can be seen as the local steepness of a diagonal when the sources are grouped by function. Furthermore, the source function can be mapped to their position in an alignment. The target addresses are indexed to close gaps in the address space, which may occur due to segmentation. To make the targets comparable between two executables, their indices are scaled and shifted.

Function Alignment The alignment technique uses Hirschberg's Algorithm to calculate an alignment of two sequences of functions. The functions are represented either by their raw length or their scaled mean-free length. Individual functions are compared using $1/(1 + |l_a - l_b|)$ as a similarity measure (1 for identical length, approaching 0 for a growing difference). This allows the alignment calculation to be free of any threshold to determine identity of two functions. Instead it maximizes the sum of similarities.

There are several other function comparison algorithms available, most notable are BinDiff [2], which compares the control-flow graphs of individual functions and the works of Stojanovic [5] and Berta [1], which calculate complex metrics on functions and then use information retrieval techniques.

2 Applications

Our techniques allow for a fast visual overview of both single executables and pairs of aligned executables. The structures in the visual overview can be interpreted by a human inspector and help to identify modules and libraries (coupling in and between modules), and highlight functions and modules with high fan-in (vertical line) or fan-out (horizontal line).

Reverse Engineering Black-Box Embedded Systems The techniques were originally developed in a reverse engineering scenario, where one or more executables from otherwise undocumented embedded systems are analyzed, to answer high level questions about the system. Such a question can for example be: What happened in the system to trigger the fault-condition 63? Often there is more than one executable of the system available, because the software was sampled from several systems with different versions.

The plot representation can be used for a first orientation in the executable, to identify so-called hotspots, such as printf, and, thus, make the disassembled code much more readable. Additionally, some of the program's modularization can already be reconstructed from the representation (see [7]). In the reverse-engineering scenario, the alignment technique useful when a second executable make the variability explicit or the first has become obsolete and the effort of reverse engineering the first needs to be preserved.

Clone-and-Own Analysis for Binaries In a scenario where code gets cloned and changed in a differ-

ent (potentially hidden) repository and just a binary gets published, the techniques can be used to show the changes made to the binary.

A manual inspection of the aligned representations allows an identification of similar executables both for license enforcement (e.g., GPL) and change inspection (analysis of modifications).

Comparing Change Logs and Binaries In a scenario where an outside contractor delivers closed source binaries and provides information about the changes made, the alignment techniques can be employed to quickly identify parts of the binary where the changes were made.

For most updates, the build process and general setup can be assumed as stable, which implies that the two executables will be very similar and all changes to the functions will be due to feature additions or bugfixes. In the alignment the function sequence will show insertions or deletions where features are added or removed, and the similarities of aligned function pairs will reflect the bugfixes.

Locating Software Quality Hotspots The last scenario is the location of hotspots to be analyzed for software quality inspection, based on the representation. It allows one to easily identify frequently used functions and check their quality properties. Additionally it is possible to identify functions that distribute the control flow over large parts of the executable. Of course, one still has to check manually whether such a distribution is actually legitimate (debug or shell interfaces) or represents a poor design.

3 Example: Malware Putty

In 2015 a malicious clone of the popular ssh client Putty was discovered [3]. The inspections showed that the main modification was the addition of a function that is called during the authentication and leaks the credentials used. The malicious executable was compiled using a newer version of Visual C++.

Figure 1 shows the application of the alignment and representation of the malicious version (left) and version 0.63 (right) of Putty. The alignment was calculated on the scaled function lengths. This allows the alignment to be invariant to the changes in the toolchain. The color of the plot points signifies the similarity of function pairs (black is 0, lightblue is 1). The changeset is shown using blue marker lines that mark functions not present in the opposite executable.

The alignment shows a visually very similar sequence of functions (diagonals), with high similarities between the functions. There are several insertions and deletions marked in the alignment, which suggests that the clone is not based on version 0.63 directly. The last part of the diagonal represents the C library, which shows huge differences due to the changed compiler versions. The malicious function is one of the functions added in the MalPutty executable.

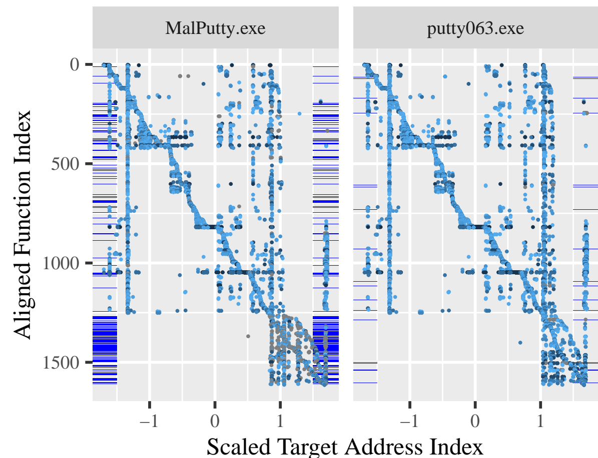

Figure 1: Alignment of a Malicious Version of Putty (left) with Putty Version 0.63 (right)

By this example, we can demonstrate the aforementioned application scenarios. First, the malicious Putty is a direct example of the clone-and-own scenario. Second, the Putty executable is relatively standalone, not depending on a lot of external libraries, and therefore represents the closedness of executables of embedded systems. Third, since it is unclear on which commit exactly the clone was forked, it can possibly also be closer to Putty 0.64. Selecting such a base executable resembles the analysis of change logs (not shown here). Lastly, the set of changed and inserted functions can be used to identify the possible set of malicious functions.

References

[1] K. Berta et al. “Estimation of similarity between functions extracted from x86 executable files”. In: *SJEE* (2015).

[2] T. Dullien et al. “Graph-based comparison of executable objects”. In: *SSTIC* (2005).

[3] C. Fry. “Trojanized PuTTY Software”. In: *Cisco Security Blog* (2015). URL: `http://blogs.cisco.com/security/trojanized-putty-software`.

[4] D. S. Hirschberg. “A Linear Space Algorithm for Computing Maximal Common Subsequences”. In: *Com. ACM* (1975).

[5] S. Stojanović et al. “Approach for estimating similarity between procedures in differently compiled binaries”. In: *IST* (2015).

[6] A. Wichmann et al. “Matching Machine-Code Functions in Executables within one Product Line via Bioinformatic Sequence Alignment”. In: *MUD* (2015).

[7] A. Wichmann et al. “Visual Analysis of Control Coupling for Executables”. In: *WSRE* (2015).