Towards Robust Instruction-Level Trace Alignment of Binary Code

Ulf Kargén and Nahid Shahmehri
Department of Computer and Information Science, Linköping University
Linköping, Sweden
{ulf.kargen, nahid.shahmehri}@liu.se

***Abstract*—Program trace alignment is the process of establishing a correspondence between dynamic instruction instances in executions of two semantically similar but syntactically different programs. In this paper we present what is, to the best of our knowledge, the first method capable of aligning realistically long execution traces of real programs. To maximize generality, our method works entirely on the machine code level, i.e. it does not require access to source code. Moreover, the method is based entirely on dynamic analysis, which avoids the many challenges associated with static analysis of binary code, and which additionally makes our approach inherently resilient to e.g. static code obfuscation. Therefore, we believe that our trace alignment method could prove to be a useful aid in many program analysis tasks, such as debugging, reverse-engineering, investigating plagiarism, and malware analysis.**

We empirically evaluate our method on 11 popular Linux programs, and show that it is capable of producing meaningful alignments in the presence of various code transformations such as optimization or obfuscation, and that it easily scales to traces with tens of millions of instructions.

I. Introduction

Recently, matching binary code has received significant attention in the literature. Several works focus on the problem of searching a large corpus of binaries for known security bugs [1], [2], [3], [4]. Applications outside security include detection of plagiarism [5], [6], [7] or code cloning [8].

Several existing methods [9], [10], [11] match binary code by means of approximate control-flow graph isomorphism, and can achieve good results in cases where small semantic changes have been applied to otherwise syntactically identical binaries. However, in cases where binaries exhibit significant syntactic differences, e.g. due to the use of different compilers or optimization settings, such methods typically fail. Therefore, *semantic* matching, based on symbolic execution [12], [13], [7], has been suggested as an alternative. While such methods can handle syntactic differences better, they are often prohibitively expensive for real-life programs. Recently, several works have tried to overcome these shortcoming by using various approximate code-similarity measures [1], [2], [4], [14], [3]. However, to provide sufficient semantic context for matching, these methods often match code at a very course granularity, typically on the level of entire functions.

Moreover, to the best of our knowledge, all existing methods can only compute *static* code mappings, i.e. they create mappings between static code segments in two binary executables. In this work, we tackle the related problem of aligning *dynamic* instruction traces. That is, we focus on creating fine-grained mappings between dynamic instruction *instances* of two program executions, a problem that, to the best of our knowledge, has not previously been treated in the literature. Being able to determine corresponding points of execution in two semantically similar but syntactically different programs can provide important information in many program-comprehension scenarios. For example, an analyst may be faced with the task of debugging or binary-patching a legacy binary (possibly compiled for another architecture or with a different compiler), and for which the source code might have been lost. In this case, she can use a more recent version of the binary and align execution traces of the two binaries. If source code and debug symbols are available for the modern version of the binary, this information can be leveraged to understand the workings of the legacy binary. We give a more detailed example of such a scenario in Section V of this paper. Another potential application area is aiding in malware analysis. Criminals commonly distribute multiple binary versions of the same malicious software, using different obfuscation methods to transform each sample, in order to thwart signature-based anti-virus software. A malware analyst can use trace alignment to identify that two malware samples behave the same, and to better understand new obfuscation schemes used by malware authors. Moreover, when analyzing two slightly semantically different executables, e.g. for studying the effects of security patches [15], dynamic trace alignment can aid in the analysis by showing how concrete *computations* differ between the two executions, not just which static instructions that are different.

In this paper we present a general approach to fine-grained binary trace alignment. Given two semantically similar (but not necessarily identical) binaries, we run both binaries with the same input, and record runtime traces of their executions. In particular, we record all concrete input and output values of instructions, which we use as our principal matching feature. Since concrete values capture information about a program's *semantics* rather than its syntax, our method is highly resilient to syntactic differences stemming from e.g. optimization.

We hope that this work will serve as a foundation for further research into trace alignment as an aid in various program-analysis tasks. In summary, the main contributions of this work are as follows:

- We present a method for aligning binary-code traces that are semantically similar, but may exhibit significant syntac-

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

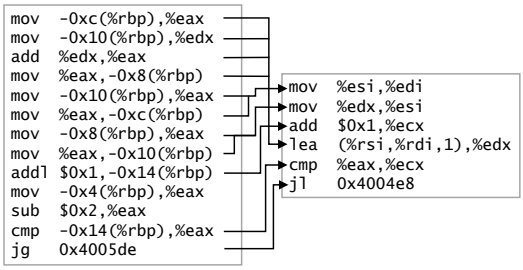


Fig. 1. Unoptimized and optimized versions of x86-64 code for computing Fibonacci numbers.

tic differences. Since our method is agnostic to the nature and origin of syntactic differences, it is able to handle trace alignment across, for example, different optimization levels, compilers, or architectures. Furthermore, since our approach is *purely dynamic*, it is inherently resilient to static code-obfuscation techniques.

- We describe a novel approach to achieving scalable trace alignment, and show that our method easily scales to traces with tens of millions of instructions.
- We empirically evaluate our method on 11 different Linux programs, and show that it can produce meaningful trace alignments for binaries compiled with different compilers, optimization levels, or target architectures, as well as binaries subjected to obfuscation.
- Finally, we demonstrate the utility of our method in a practical use case: Reverse-engineering of legacy code for binary patching.

II. Overview

Aligning code traces compiled with e.g. different optimization settings is a challenging problem, since a perfect mapping between instructions is often not possible. Moreover, complete sequential consistency between traces typically cannot be assumed. Consider the two code snippets in Figure 1. Both blocks of code represent the main computation loop in a simple program for calculating Fibonacci numbers. Both versions have been compiled with `gcc` for the x86-64 architecture, but the code to the left has been compiled without optimization, and the code to the right has been compiled with optimization level `O3`. The arrows between the code blocks represent an approximate manual mapping between instructions. Due to arithmetic simplifications performed by the compiler, the ordering of some of the (roughly) equivalent instructions is not preserved across the two versions. Also, some instructions in the unoptimized code have no clear counterpart in the optimized version.

A. Dynamic Time Warping

While semantics-preserving code transformations, such as optimization, will introduce small local discrepancies between traces, semantic differences when aligning across different revisions of a program may introduce significant large-scale discrepancies. A generic trace alignment method must therefore adopt a pragmatic approach, performing a best-effort global alignment, while tolerating local unalignable sections of traces. *Dynamic time warping* (DTW) is a well-known method for aligning time series of analogue signals, used extensively in e.g. speech recognition [16]. Given two time series $\mathbf{x}, \mathbf{y}$ of respective lengths n and m, and a distance measure $dist(x_i, y_j)$ of individual elements in the time series, a *cost matrix* C is constructed. Each cell $c_{i,j}$ represents the dissimilarity (or cost) $dist(x_i, y_j)$ between the corresponding elements in the two time series. DTW constructs a *warping matrix* W, where each element $w_{i,j}$ contains the accumulated cost of the optimal (cost minimizing) path in C from $c_{0,0}$ to $c_{i,j}$. The final minimum accumulated cost can be retrieved from element $w_{n,m}$ in the warping matrix, and is called the *warping distance*. When W has been constructed, finding the optimal alignment between $\mathbf{x}$ and $\mathbf{y}$ simply entails a greedy search through W from $w_{n,m}$ to $w_{0,0}$, to find the global minimum-cost path, or *warp path*. The row and column of each point on the warp path give the actual element alignments. For example, if the warp path passes through cell $w_{i,j}$, then elements x_i and y_j align with each other. A vertical or horizontal section of the warp path means that one element in one of the time series aligns with several elements in the other series. Figure 2 shows a 3x5 matrix with a possible warp path (thick dark line). In this example, elements 1, 2, and 3 of the vertical series align respectively with elements (1,2), (3,4), and 5 of the horizontal series.

The classical dynamic programming DTW algorithm has time complexity $\mathcal{O}(nm)$, which unfortunately limits its use to fairly short sequences. The *FastDTW* algorithm [17] is a popular approximate version of DTW with linear time complexity. It works by aggregating adjacent elements in each time series to create a low-resolution version of the cost matrix, and then running standard DTW on the low-resolution matrix. The final warp path is created by iteratively repeating this process at higher and higher resolutions, but restricting the construction of the warping matrix to a narrow band around the warp path from the previous iteration. The width of the band is controlled by a *radius* parameter, which determines how many additional elements on each side of the previous warp path to include.

B. Aligning Instruction Traces

The application of DTW for real-valued signals is straightforward. The euclidean distances between elements can be used to construct the cost matrix, and averages over several adjacent elements in a time series can be used to construct lower-resolution versions of the problem for FastDTW. However, in our setting the elements of the time series are machine code instructions. One option would be to treat individual opcodes as symbols in an alphabet, and then compute the string edit distance (essentially a variant of DTW for strings). However, such an approach would restrict us to align code generated by the same compiler using the exact same compilation settings, since, for example, optimized binaries often use more efficient instructions to perform the same computations as their unoptimized counterparts. Instead, we record the concrete input

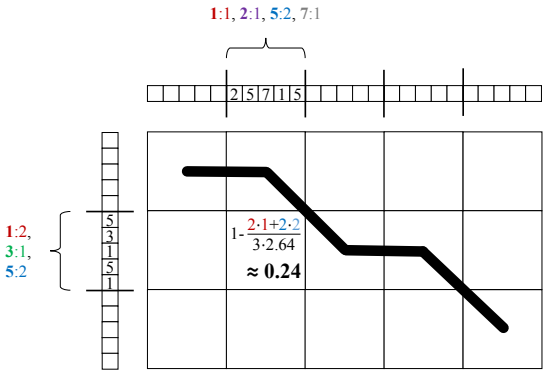


Fig. 2. Constructing a cost matrix using a vector space model of instruction segments.

and output *values* of computations, and use these to match instructions. This approach also facilitates a natural way to aggregate several elements of a trace. A simple approach is to represent a sequence of several instructions, henceforth called a *segment*, as the set of all unique values observed in that segment, and use the Jaccard similarity to compute distances between segments.

A limitation of the Jaccard similarity, however, is that it does not account for the frequency distributions of values. Some values, such as the value '0', or small powers of two, are extremely common in most program traces, and may incorrectly inflate matching scores. Also, since several segments may contain similar sets of values, it is also beneficial to consider the *number of times* a certain value is observed within a segment. To this end, we adopt the approach proposed by Kargén et al. [18], utilizing the *vector space model* with tf-idf weighting from the field of information retrieval. When the vector space model is used for information retrieval, each document in a collection is represented by a point in a multidimensional space, with one dimension per unique term in the vocabulary of the document collection. The "coordinates" of a document are determined by the number of times each term occurs in that document, denoted as the *term frequency* (tf). The tf components are also weighted by the *inverse document frequency* (idf) of the corresponding term, to create the final tf-idf vector components. The idf of term t is calculated as $\log(N/n_t)$, where N denotes the total number of documents, and n_t denotes the number of documents in which term t occurs at least once. The rationale for the idf factor is that common terms with low discriminative value receive lower weight in the document vectors. Finally, the similarity of documents can be computed as a value between 0 and 1 by calculating the *cosine similarity* between document vectors, defined simply as the cosine of the vectors, i.e.

$$\text{similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$$

The cost (or distance) is then given by $1 - \text{similarity}(\mathbf{a}, \mathbf{b})$

In our setting, documents correspond to trace segments, and terms correspond to values. We apply the tf-idf technique to construct vector-space representations of trace segments, henceforth referred to as *value vectors*. Figure 2 shows a toy example of this approach, where the aligned traces are 15 and 25 elements long, respectively. For simplicity, we assume that each element has a single value, and that idf-weighting is not used. A segment size of 5 is used, resulting in a 3x5 cost matrix. The highlighted segments show how value vectors are constructed. For example, the vertical segment has two occurrences of the value '1', one occurrence of '3', and two occurrences of '5'. When computing the cosine of the two value vectors, only the shared components, corresponding to values '1' and '5', contribute to the dot product. After dividing by the vector lengths and subtracting from 1 to get a distance metric, we arrive at a cosine distance of about 0.24.

In the following section, we describe in more detail how values are recorded into traces, and our adaption of the FastDTW algorithm for trace alignment.

III. Design and Implementation

Our system consists of four modular components that are executed in a pipeline. The architecture-specific *trace recorder* component collects detailed syntactic instruction traces of executions. The *trace distiller* takes an instruction trace from the trace recorder and produces an architecture-agnostic *value trace* consisting only of concrete values observed for each instruction. A *trace filterer* component can optionally be applied to the value trace to remove irrelevant parts of the execution, such as instructions in external libraries. Finally, the *trace aligner* component takes two value traces, produced by the preceding steps of the pipeline, and performs trace alignment according to the procedure in Section II.

We implemented our method using about 4,200 SLOC C++, and a few hundred lines of Python for parsing results, calculating statistics, etc. Below, we describe each component of our system in more detail.

A. Trace Construction

Trace recording. Our trace recording is implemented using the Pin dynamic binary instrumentation (DBI) framework [19]. All read and written register and memory locations, including concrete addresses, are recorded, along with the values of all input and output operands.

Trace distilling. The trace distiller strips a trace from the trace recorder of all syntactic information, and creates an architecture-independent value trace. Entries in the value trace consist only of a unique instruction ID, and a set of input and output values. The trace distiller also outputs a mapping between instruction IDs and actual instruction offsets in executables.

Inputs and outputs are xor-ed with their own respective magic constant, so that e.g. the value '3' as an input is distinct from the value '3' as an output. This way, we capture more information about the input/output semantics of instructions. To allow alignment across e.g. 32/64-bit architectures, and to handle optimized code that performs calculations in parallel on several values using SIMD-instructions, we *tokenize* all values to a common bit-width. Values greater than the token-width are split into several values, while values smaller than the

token-width are zero-padded. We currently use a token width of 32 bits, which appears to work well in practice, and allows alignment of 64 and 32-bit traces.

We also attempt to filter out addresses from the value traces. Addresses are often used as data in binaries, e.g. when passing around pointers in the code. Since addresses are typically not comparable across different versions of a binary, these unnecessarily inflate the size of traces, and add noise to the segment matching. Since a handful of false positives or negatives during address filtering will not have a detrimental effect on segment matching, we adopt a simple yet effective filtering approach: For all addresses reported by the trace recorder, we mask off the least significant 12 bits (corresponding to one 4kB page), and store it in a set of *address masks*. Every recorded value is checked against these address masks before being stored in the value trace. We found that address filtering reduced the number of stored values in traces by roughly 25%.

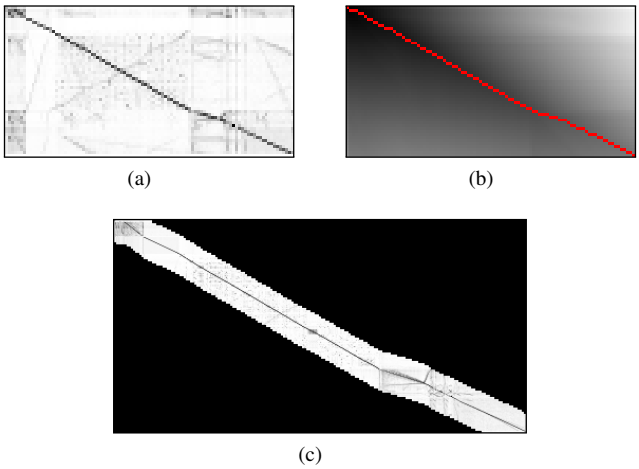
Trace filtering. It is often desirable to be able to filter out irrelevant parts of an execution trace, such as library code that is not interesting to the analysis. Such filtering will both reduce the time for trace alignment, and can improve precision by e.g. removing non-deterministic parts of an execution, which cannot be matched using value-comparisons. Our system allows filtering based on individual instruction addresses, ranges of addresses, and names of entire executables (e.g. shared libraries).

B. Trace Alignment

Warping matrix construction. The trace alignment component takes two value traces, splits them into equally-sized segments, and constructs value vectors out of these segments. The resulting arrays of value vectors are used to compute the cosine distances between segments, and construct the warping matrix. Since some instructions, such as branches or address-manipulation instructions, will not have any recorded values, a value vector is allowed to be empty. We treat empty vectors specially: If two empty vectors are compared to each other, they are considered equal, and are assigned distance 0. If one of the vectors is instead non-empty, they are considered maximally distant, i.e. with distance 1.

FastDTW implementation. The initial segment size is chosen so that the smaller dimension of the warping matrix is always 50 elements. For each iteration of FastDTW we decrease the segment size with a factor two, doubling the row and column count of the warping matrix. FastDTW terminates when we have reached a segment size of 1 for any of the traces. We use a FastDTW radius parameter of 20 elements. This means that the maximum space required for the final warping matrix is always bounded to $2 \cdot 20 \cdot n$, where n is the size of the longer of the two traces.

We have implemented a simple multithreaded variant of FastDTW, where matrix rows are processed (partially) in parallel in a pipelined fashion. When using 4 cores, we observe roughly a 3x speedup with our parallel implementation.



(a) (b) (c)

Fig. 3. First-iteration cost (a) and warp (b) matrices, and third-iteration cost matrix (c) of aligning optimized and unoptimized `gzip` traces.

Value-vector construction. We use a simple sparse array implementation based on the C++ STL `map` to store value vectors, so that values with a zero tf-idf component need not be physically stored. Also, only values that are present at least once in both traces get physical entries in the sparse arrays. Since values unique to one trace can never contribute to the dot product in the cosine-distance computation, they can be safely omitted. They still, however, contribute to the length of the vector in the denominator of the cosine. We observed that this simple optimization often massively reduced the size of value vectors.

While sparse arrays are necessary to reduce memory use, they unfortunately result in an $\mathcal{O}(n \log n)$ time complexity in the size of value vectors when computing cosine distance. This brings the overall time complexity of our FastDTW implementation up to $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n)$. To avoid this, we set an upper bound on the size of value vectors, by only keeping the 300 largest components. We found that this significantly reduces the time needed for the first coarse-grained alignment iterations of FastDTW, while not impacting the accuracy of the final fine-grained alignments (see Section IV-C).

Subsequence alignment. We have also implemented subsequence alignment [16]. In this variant of DTW, the warp path is not constrained to start and end at opposite diagonal ends of the warping matrix, but can run from any column of the top row to any column of the bottom row. Subsequence alignment is useful when a trace can only be aligned with a subsequence of the other trace. For example, given two versions A and B of a program, an analyst might be interested in aligning only code within a certain function of A with a full trace of B, in order to locate the corresponding function's instructions in B. Subsequence alignment may be necessary if only parts of one trace are comparable to another trace.

Visualization. We have implemented a tool for rendering the cost and warp matrix at each FastDTW iteration to an image file. Figure 3a shows the cost matrix of aligning traces of

an optimized (O3) and unoptimized (O0) version of gzip. Each pixel of the 50x98 matrix represents 7738 instructions. The aspect ratio of about 2:1 is due to the optimized code using roughly half as many instructions to perform the same computations as the unoptimized code. Darker shades in the matrix indicate lower cost (i.e. lower dissimilarity). We can see a clear diagonal line of dark pixels, representing corresponding segments of instructions in both traces. We can also discern several symmetrical square regions along the diagonal, indicating different distinct stages of processing. Figure 3b shows the corresponding warping matrix, with the optimal warp path highlighted in red. We see that the warp path closely follows the dark diagonal line in the cost matrix. Figure 3c shows the third-iteration cost matrix of FastDTW for the same alignment problem. With 4 times higher resolution, finer details of the trace similarities are now visible. Since we use a radius parameter of 20, most of the cost matrix is never constructed. Ignored regions are shown as black pixels here.

IV. Empirical Evaluation

In this section we empirically evaluate the effectiveness of our trace alignment method. There are several challenges with such an evaluation. First, since we are, as far as we know, the first ones to consider the problem of dynamic trace alignment, direct comparisons against other work is not possible. A second problem is that ground truth is difficult to obtain for evaluating our method's effectiveness when aligning programs that are not semantically identical. As a first step towards evaluating the feasibility of our approach, we focus on measuring the robustness of our method against various semantics-preserving code transformations. In section V we then present a practical use-case, showing the utility of our method when comparing two revisions of the same program.

The programs used in our experiments, and a brief description of the respective inputs used, are shown in Table I. Except when explicitly stated otherwise, the compiler used to generate all binaries was gcc 4.8.5. All binaries were compiled with embedded debug info as ground truth. Experiments were performed on a Linux Mint 17.2 workstation equipped with a quad-core 3.3 GHz Intel Xeon E3-1245 CPU with hyperthreading, and 16 GB RAM.

External dependencies, such as the C library, constitute an error source in our experiments. Within an instruction trace, code from external shared libraries will be identical for all versions of the same program, and would therefore be trivial to align. This may "aid" the DTW algorithm in finding an accurate global alignment for the entire trace. To accurately characterize our method's accuracy under maximally adversarial conditions, we used the trace filterer (Section III-A) to remove instructions from external libraries, but kept instructions from libraries that programs ship with. For example, for the xmllint program, we removed instructions in libc, but kept instructions in libxml. The last three columns of Table I show some execution details of the unoptimized (gcc O0) versions of binaries. The columns show, respectively, the number of dynamic instructions, the number of dynamic instructions after trace filtering, and the number of unique instructions in the filtered trace.

A. Metrics Used

In a typical usage scenario, an analyst may be interested in aligning a *reference* binary, for example with embedded debug info, against a *target* binary, which may e.g. be stripped, to gain insights about the target binary. In our experiments, we used reference binaries compiled with gcc using optimization level O0, and aligned against traces from binaries subjected to various code transformations.

To evaluate the accuracy of our alignment method, we compute the *alignment delta* Δ_a of trace alignments using embedded debug info. Consider two program versions X and Y. Given an alignment between traces from the two versions, let element x_i denote the i:th element of X's trace, and y_j the j:th element of Y's trace. If x_i align with y_j, then the Δ_a of that particular instruction alignment is $|j - k|$, where k is the index closest to j such that x_i maps to the same source code line as y_k. If x_i aligns with several instructions $[y_j, ..., y_{j+l}]$ in the other trace, then Δ_a is 0 if any of those instructions map to the same source code line. Otherwise, Δ_a is calculated as $\min(|j - k|, |(j + l) - k|)$, where k is the matching index closest to either of y_j or y_{j+l}. Thus, Δ_a measures how close an individual instruction alignment is to the closest "equivalent" instruction in the target trace.

The main contribution of our method is the ability to produce dynamic instruction alignments. However, in order to allow an, at least partial, comparison with future and present code-matching methods, we also studied our method's ability to produce *static* instruction mappings. For each static instruction in the reference binary, we iterate over all its dynamic instances in the reference trace, and record the instructions with which it aligns in the target trace. From this, we collect statistics on how many times each instruction in the reference binary aligns with each instruction in the target binary. For a given instruction in the reference binary, we can then create a list of most likely corresponding instructions in the target binary, ranked by the number of alignments. The accuracy of static mappings are evaluated using embedded debug info, where instructions in the respective binaries that map to the same source code line are considered to match.

In the following experiments (Section IV-B), we report the fraction of static instruction mappings where the topmost candidate is the correct match. However, while our method finds exact matches for many instructions, a perfect static instruction mapping between binaries is difficult to achieve due to e.g. code layout differences, as discussed in Section II. Therefore, it is also interesting to investigate our method's ability to pinpoint *regions* of binaries where the same high-level functionality is implemented, or to identify corresponding execution points where the same high-level operations are performed. We therefore propose an additional metric that better capture our method's ability to perform fine-grained, but not necessarily exact, trace alignment. Specifically, we evaluate how well trace alignment performs in finding instruction that

TABLE I
PROGRAMS USED IN THE EXPERIMENTS.

Program	Version	Input/execution details	Dyn. instr. (O0)	Filtered instr.	Static instr.
bzip2	1.0.6	Compress 3.8 kB text file file to standard output	7,227,631	7,048,780	5,971
date	8.26	Invoked without additional command-line options	242,535	2,725	950
df	8.26	Invoked without additional command-line options	794,276	97,127	3,335
djpeg	9b	Decode 4.1 kB JPEG file to standard output	2,097,285	1,446,552	9,891
gzip	1.8	Same as bzip2	873,672	753,949	3,740
lighttpd	1.4.45	Server startup + HTTP GET request for 84 B HTML file	2,314,350	573,108	17,898
ls	8.26	Invoked with 'ls -l /'	1,063,013	151,724	5,275
mpg123	1.23.8	Decode 8.0 kB MP3 file to standard output	10,274,764	6,020,323	15,976
sha256sum	8.26	Checksum text file (same as bzip2) to standard output	539,236	289,073	5,397
sqlite	201701170010	Dump 9.2 kB database to standard output as SQL	3,376,993	2,255,913	42,378
xmllint	2.9.4	Run on 17.7 kB XML file with the --noout option	1,746,810	1,061,571	12,927

TABLE II
INSTRUCTION MATCHING ACCURACY, GCC O0 VERSUS GCC O3

Program	$\Delta_a = 0$	$\Delta_a < 10$	Top	Close
bzip2	73.00%	98.81%	47.98%	79.70%
date	61.69%	95.38%	50.00%	72.95%
df	44.33%	84.79%	40.72%	63.87%
djpeg	64.20%	96.54%	47.30%	77.56%
gzip	68.25%	97.69%	54.30%	82.11%
lighttpd	50.59%	94.59%	39.61%	63.74%
ls	56.88%	91.28%	41.63%	67.68%
mpg123	60.77%	95.17%	44.20%	69.25%
sha256sum	77.89%	93.46%	69.59%	94.89%
sqlite	51.25%	94.55%	36.29%	61.83%
xmllint	56.29%	92.13%	41.46%	71.93%

TABLE III
INSTRUCTION MATCHING ACCURACY, GCC O2 VERSUS GCC O3

Program	$\Delta_a = 0$	$\Delta_a < 10$	Top	Close
bzip2	93.89%	99.56%	86.89%	91.98%
date	96.40%	98.52%	94.12%	95.45%
df	89.47%	98.20%	88.86%	94.09%
djpeg	98.48%	99.03%	95.74%	98.45%
gzip	95.09%	99.35%	87.02%	95.33%
lighttpd	93.31%	97.67%	86.09%	89.70%
ls	90.78%	98.14%	87.72%	92.33%
mpg123	94.60%	99.53%	89.88%	95.60%
sha256sum	97.43%	99.02%	97.01%	99.31%
sqlite	89.78%	97.93%	80.51%	87.54%
xmllint	96.27%	99.49%	89.59%	94.26%

are *physically close*, but not necessarily identical, to the exact match. To this end, we measure the number of cases where the source-code line of the topmost mapping is less than 10 lines away from the correct match.

B. Alignment Accuracy

Here, we evaluate how the alignment accuracy of our method is affected by a number of code transformations.

Aligning optimized code. In our first experiment, we aligned optimized and unoptimized x86-64 binaries compiled with `gcc`. In this experiment, we used reference binaries compiled with optimization level `O0`, and `O3`-optimized target binaries.

Table II shows accuracy metrics for the programs in the experiment. The first two columns show, respectively, the percentages of dynamic instruction alignments with $\Delta_a = 0$, i.e. "exact" matches, and $\Delta_a < 10$, i.e. temporally close alignments. With the exception of `df`, more than 50% of alignments are exact for all programs, according to debug info, and more than 90% of alignments are close to an exact match.

The two rightmost columns show results for static instruction mappings. The column *Top* shows the fraction of instructions in the reference (`O0`) binary where the topmost entry in the ranked list of static mappings is the correct match. We see that, for most programs, the top ranking match is correct in about 50% of cases. The percentage of cases where the topmost mapping is close to the true match in the source code is shown in the column *Close*. These figures vary between 62% and 95%, showing that our method can be used to create approximate static mappings for large portions of executed instructions.

For reference, we also performed the same experiment using `O2`-compiled binaries to generate reference traces. The results are shown in Table III. With the highly similar `O2` and `O3` optimization levels, percentages are very high across all accuracy statistics.

Aligning x86 and x86-64 code. To study the performance of our method when aligning across 32 and 64-bit architectures, we repeated the above experiment, aligning `O0` and `O3` `gcc` traces, but this time compiling the target binaries for the 32-bit x86 architecture. Results are shown in Table IV. Precision figures are similar to the same-architecture case, but generally a few percentage points lower, showing that our method can tolerate syntactic differences between x86 and x86-64 code.

Aligning across different compilers. In the next experiment, we compiled the target binaries with `clang` 3.5 and optimization level `O3`. Both sets of binaries were compiled for x86-64. As can be seen from Table V, results are generally comparable to those of the same-compiler experiment, with some programs even getting higher accuracy scores. The exception is the `date` program, where results are significantly lower for static mappings. This is likely due to its very short trace length. Since most of the work in `date` is carried out in external libraries, the filtered trace is only a few thousand instructions long. Moreover, most instructions in the trace execute only once, making static mappings very sensitive to small misalignments.

TABLE IV
INSTRUCTION MATCHING ACCURACY,
X86-64 GCC O0 VERSUS X86 GCC O3

Program	$\Delta_a = 0$	$\Delta_a < 10$	Top	Close
bzip2	69.69%	99.03%	48.17%	77.42%
date	52.26%	82.90%	42.32%	56.84%
df	41.23%	78.01%	35.68%	56.40%
djpeg	68.99%	98.42%	49.90%	78.73%
gzip	66.44%	99.39%	56.10%	82.11%
lighttpd	54.02%	94.64%	40.73%	64.91%
ls	51.44%	85.40%	36.25%	57.38%
mpg123	28.05%	48.37%	38.44%	60.43%
sha256sum	70.46%	89.30%	62.96%	93.14%
sqlite	50.83%	91.78%	36.33%	59.51%
xmllint	57.20%	90.44%	39.28%	66.27%

TABLE V
INSTRUCTION MATCHING ACCURACY,
GCC O0 VERSUS CLANG O3

Program	$\Delta_a = 0$	$\Delta_a < 10$	Top	Close
bzip2	65.50%	97.95%	47.66%	76.97%
date	45.72%	99.16%	23.26%	34.53%
df	50.20%	93.60%	43.54%	70.82%
djpeg	70.85%	94.40%	46.91%	77.12%
gzip	76.39%	98.14%	59.63%	85.56%
lighttpd	61.38%	96.40%	40.70%	65.66%
ls	54.11%	92.38%	40.63%	67.07%
mpg123	61.06%	96.15%	48.93%	75.35%
sha256sum	97.61%	99.99%	87.75%	94.77%
sqlite	50.63%	91.88%	36.88%	62.16%
xmllint	61.58%	94.15%	42.28%	74.56%

Aligning obfuscated code To evaluate the accuracy of our method under adversarial conditions, we also performed trace alignment of obfuscated binaries. We used the open-source Obfuscator-LLVM (`ollvm`) [20], which is implemented on top of the LLVM framework, and which applies obfuscation during compilation. `ollvm` was chosen in part because its design allows embedding debug info as ground truth also in obfuscated binaries. It should be noted, however, that the accuracy of debug info in obfuscated code is far from perfect, due to the heavy code transformations. As such, the results in this section should be seen as indicative, rather than providing exact figures of our method's performance on obfuscated code.

Three types of obfuscations are supported by `ollvm`: *bogus control flow*, *instruction substitution*, and *control flow flattening*. Bogus control flow inserts fake basic blocks guarded by opaque predicates, to complicate static analysis of control-flow. Instruction substitution transforms computations by replacing arithmetic operations with more convoluted, but semantically equivalent, sequences of computations. The latter is a more significant challenge for trace alignment, since traces of obfuscated code will be "polluted" with intermediate values not present in the non-obfuscated traces.

Control flow flattening hinders static control-flow recovery by replacing all direct branches with indirect jumps, routing all control flow through a dispatch loop. The open-source variant of `ollvm` applies this obfuscation very aggressively, flattening control flow between every basic block in the program. This results in both significant slowdowns and code-blowup. Unfortunately, debug info for the inserted flattening-code also appears to map instructions to source-code lines in a somewhat arbitrary way, leading to difficulties in evaluating the accuracy of trace alignment when this obfuscation is used. We therefore only used bogus control flow and instruction substitution in our experiments. All `ollvm`-obfuscated binaries were built for the x86-64 architecture.

TABLE VI
INSTRUCTION MATCHING ACCURACY,
GCC O0 VERSUS OLLVM

Program	$\Delta_a = 0$	$\Delta_a < 10$	Top	Close
bzip2	74.27%	81.24%	44.85%	63.31%
date	6.97%	22.13%	6.00%	11.37%
df	29.71%	43.15%	34.57%	50.70%
djpeg	64.06%	72.22%	37.94%	56.41%
gzip	65.38%	72.99%	47.03%	68.13%
lighttpd	14.75%	24.97%	12.93%	23.83%
ls	16.04%	21.18%	18.12%	29.48%
mpg123	58.06%	71.68%	28.36%	48.24%
sha256sum	97.58%	98.43%	88.03%	91.59%
sqlite	32.59%	40.79%	25.53%	37.45%
xmllint	27.95%	34.98%	16.96%	31.85%

Results for the different programs are shown in Table VI. While the accuracy is still reasonable for several programs, it suffers considerably for others. As in the previous experiment, the short trace length of `date` becomes problematic. The very low percentages for Δ_a indicate that DTW has found a degenerate alignment for `date`. Similarly, other programs with relatively short trace lengths compared to the number of executed static instructions (`lighttpd`, `ls`, `sqlite`, `xmllint`) suffer reduced static mapping accuracy. Again, it should be noted that the reduced quality of debug information in obfuscated binaries may also contribute to the drop in accuracy.

Sources of imprecision. Even under heavy syntactic transformations, our method is generally able to produce global alignments where individual instruction instances in the reference trace are aligned with instructions that are *close* to their true counterpart in the target trace, as indicated by the high $\Delta_a < 10$ figures. However, in many cases instructions are not precisely aligned, as indicated by the markedly lower $\Delta_a = 0$ figures. One reason for this is that many instructions do not perform computations useful for value-comparisons. For example, on average 37% of instructions in the `O0` traces had empty value-sets. Also, the frequency distribution of values in program traces is highly skewed, as discussed in section II-B. During the first FastDTW iterations (with large instruction segments), tf-idf weighting efficiently compensates for this skew, but individual instruction alignments during the last FastDTW iteration still suffers from reduced accuracy. Finally, we again note that precise instruction alignments are not always possible, as shown in the example in Figure 1.

C. Scalability

Scalability is an important factor when applying trace alignment to real-life problems. In this section, we have

TABLE VII
ALIGNMENT TIMES IN SECONDS, WITH VALUE VECTOR BOUNDING ENABLED (LEFT COLUMNS) AND DISABLED (RIGHT COLUMNS)

Program	O0/O3		O0/clang		O0/x86		O0/ollvm	
bzip2	168.37	651.76	154.63	520.32	154.50	522.60	394.41	869.48
date	0.13	0.12	0.10	0.19	0.06	0.07	0.16	0.21
df	1.25	1.14	1.17	1.11	1.27	1.14	3.80	3.68
djpeg	57.37	287.46	55.75	263.45	55.51	255.40	133.89	405.29
gzip	17.72	29.60	17.57	29.44	17.49	29.67	52.73	68.45
lighttpd	5.65	5.49	5.68	5.50	5.60	5.57	28.99	27.00
ls	2.03	1.79	1.96	1.75	2.35	2.10	7.75	7.13
mpg123	503.05	4816.31	501.67	4660.36	361.40	1370.62	548.52	4875.72
sha256sum	22.61	91.89	23.45	89.54	23.25	92.72	86.32	189.72
sqlite	29.27	29.88	29.50	29.28	28.50	29.05	124.50	120.51
xmllint	13.08	12.98	12.85	12.67	13.07	12.94	60.24	56.84

evaluated the efficiency of our method for different programs and trace sizes. In the below experiments, alignments were performed using 8 parallel threads (i.e. one thread per virtual CPU core).

We first study the impact of our value vector bounding approximation (Section III-B). Table VII shows alignment times for the experiments in Section IV-B (the `O2` versus `O3` case has been omitted to save space). The left and right columns for each experiment show alignment times with and without value vector bounding, respectively. With the optimization active, all alignments finish within 10 minutes, with the majority taking less than a minute. While the value vector bounding has little impact for programs with short trace lengths, it improves alignment times with a factor 10 or more for the `mpg123` program. Interestingly, `mpg123` also has significantly longer alignment times than `bzip2`, despite the two having traces of comparable size. The difference is likely due to `mpg123` doing more heavy numerical computations, with more runtime-values per instruction on average than `bzip2`.

To ascertain that value vector bounding does not negatively affect alignment accuracy, we also compared the final warp paths from both sets of experiments, and found that they were identical for all programs. Thus, value vector bounding does not appear to negatively effect the accuracy of final alignments.

To evaluate the scalability when using longer traces with tens of millions of instructions, we selected four of the more computationally heavy programs used earlier, and repeated the `gcc O0` versus `gcc O3` alignment, using similar but larger inputs. The size of inputs and resulting filtered trace sizes (in millions of instructions) are shown in the first columns of Table VIII. The two rightmost columns show respectively the time to prepare (i.e. record, process, and filter) traces, and the time to compute alignments. The alignment times are roughly consistent with the theoretical linear time complexity of FastDTW.

TABLE VIII
ALIGNMENT TIMES FOR LONG TRACES.

Program	Input (kB)	Trace sizes (millions)	Prep. (s)	Align (s)
bzip2	93.8	49.1 x 19.1	14.8	1512.0
djpeg	61.1	72.1 x 35.9	35.6	2080.9
mpg123	63.7	32.1 x 18.3	19.4	3310.9
sqlite	884.7	126.7 x 62.4	70.9	3427.9

V. CASE STUDY

In this section we demonstrate the utility of our method in a practical use case. In our scenario, we have a legacy binary, for which it is unfeasible to recompile from source code, e.g. because the source code has been lost. We also have a more modern version of the binary, where source code is available. If a bug is discovered and patched in the modern version, we may need to binary-patch the legacy version if it is still in use in some production environment, and cannot be upgraded due to e.g. backwards compatibility issues. As an example of how trace alignment can be used in such a scenario, we use two versions of the `libtiff` library, and the bundled `tiff2pdf` tool. `libtiff` version 4.0.3 has a memory corruption bug (CVE-2014-8129), which could potentially be exploited by an attacker by using a specially crafted TIFF-file as input to `tiff2pdf`. The bug is patched in subsequent versions of `libtiff`. Figure 4 shows how the bug was patched. After the patch, the function `TIFFInitNeXT` in `tif_next.c` sets a function pointer to an added input-validation routine, so that this routine is executed before further parsing of the TIFF image. In a binary-patching scenario, a straightforward approach to fix the vulnerable code would be to patch in an equivalent validation routine somewhere in the binary, and also patch `TIFFInitNeXT` to insert a pointer to this routine in the `TIFF` struct.

We now show how trace alignment can be used to aid in locating the function `TIFFInitNeXT` in the legacy binary, using the (at the time of writing) most recent 4.0.7 version of `libtiff` as reference. To make the use-case more realistic, we compiled the modern version with `gcc` for x86-64, using optimization level `O0`, and the vulnerable version with `clang` for x86, using optimization level `O3`. We filtered out external library code, as in previous experiments, and aligned executions of the respective binaries with an input that exercised the code in `TIFFInitNeXT`.

Note that this is a challenging use-case for our method, since no actual computations are made in the code of interest, and thus no runtime-values are available for matching instructions. Our method therefore needs to rely on accurately aligning

```
142. int
143. TIFFInitNeXT(TIFF* tif, int scheme)
144. {
145.     (void) scheme;
146.     tif->tif_decoderow = NeXTDecode;
147.     tif->tif_decodestrip = NeXTDecode;
148.     tif->tif_decodetile = NeXTDecode;
149.     return (1);
150. }
```

```
170. int
171. TIFFInitNeXT(TIFF* tif, int scheme)
172. {
173.     (void) scheme;
174.     tif->tif_predecode = NeXTPreDecode;
175.     tif->tif_decoderow = NeXTDecode;
176.     tif->tif_decodestrip = NeXTDecode;
177.     tif->tif_decodetile = NeXTDecode;
178.     return (1);
179. }
```

Fig. 4. Version 4.0.3 (top) and 4.0.7 (bottom) of `TIFFInitNeXT` in `libtiff`.

the surrounding context. The code in question would also be difficult to locate for existing static code matching approaches, since they typically require larger sections of code for accurate matching. For example, state-of-the-art tools like Genius [4] or DiscovRE [2] require functions to have at least five basic blocks to be searchable.

We used embedded debug info to study how well our method would perform in this scenario. The identified instruction mappings, translated to source-code lines, are shown below:

```
tif_next.c:174 → tif_next.c:147
tif_next.c:175 → tif_next.c:148
tif_next.c:179 → tif_next.c:148
tif_next.c:176 → tif_next.c:149
tif_next.c:177 → tif_dir.c:227
tif_next.c:179 → tif_dir.c:227
```

Despite about four years worth of code revisions between the versions, and despite the significant syntactic differences, our method manages to pinpoint the location where the patch should be applied within a few instructions.

VI. Related Work

In this section we briefly survey related work. Due to the large corpus of work on binary analysis, we limit our discussion to the works that are most similar to ours. Specifically, we focus on methods capable of comparing binary code without the aid of source code, and in the presence of syntactic differences between binaries.

Dynamic methods. The work most closely related to ours is the method by Zhang and Gupta [21], which were later extended by Nagarajan et al. [22]. The purpose of these works are to compute *static* mappings between instructions in two syntactically different but semantically identical binaries. Like our method, they also rely on dynamic analysis. Due to the similarities to our work, we will discuss their respective approaches in more detail.

Zhang and Gupta compare sets of runtime values to match instructions. If one instruction's value-set is a subset of the other's, the instructions are considered to match. Matches are further pruned by considering data-flow relations, using heuristic matching of the dynamic data dependence graphs of the binaries. The method is very effective at finding true matches (95% accuracy on average), but also produces many false matches (5–40% according to an experiment in the paper).

The primary purpose of Zhang and Gupta's work was to create mappings between optimized and unoptimized code. Nagarajan et al. extended their method to cases where the mapping between functions in the two binaries is not known a priori. First, the dynamic call graphs of the respective executions were aligned using a heuristic method. Zhang and Gupta's original instruction-matching method was then applied to pairs of matched functions between the binaries. Finally, the instruction mappings were used to match paths in the dynamic control-flow graphs of the two binaries. To deal with false matches, matched paths were further prioritized using the structure of the dynamic CFGs.

While both works share similarities with our work, they are also different in several important ways. First, even though they use dynamic analysis, they both compute static code mappings, while we focus on the alignment of dynamic instruction instances. Second, both methods are explicitly targeted towards matching semantically identical binaries. Indeed, Zhang and Gupta pointed out that a single differing data-flow edge in one of the two compared binaries resulted in a sharp increase in the number of unmatchable instructions, and proposed that this property could be used to detect compiler bugs. Lastly, while both works use runtime values for matching, just like our work, we use the (approximate) temporal ordering as our second principal matching feature, while they use approximate matching of the data-flow and/or control-flow graphs. Furthermore, since both their respective works rely on several "stacked" heuristics, the risk of various unknown failure modes is intuitively higher. We instead take a more fundamental approach, using only the temporal ordering and concrete computations of instructions for our matching. Similar to the bias-variance tradeoff in statistics, the price for this generality is a somewhat lower accuracy. Since the purpose of this work was to investigate the accuracy of value-based trace alignment in its own right, we refrained from integrating any ad hoc heuristics into our method. However, combining our approach with methods based on data and control flow information could be an interesting direction for future work. For example, one alternative could be to combine our method's ability to find meaningful global alignments with a local data-flow matching approach similar to Zhang and Gupta's work.

Other approaches to dynamic code matching include Egele et al. [23], who proposed *blanket execution* for binary function matching. Like our method, they also use dynamic analysis and collect runtime values for code matching. However, since their goal is to statically match queries against all functions in a binary, they execute all functions of a given executable in a randomized memory environment, and coerce some code paths to reach 100% code coverage. Jhi et al. [5] also proposed using runtime-observed values for plagiarism detection on binaries.

Their method, however, only performs matching on the level of entire binaries. Zhang et al. [6] later extended their approach for detection of algorithm plagiarism. Kirat et al. [24] proposed a method for finding the evasion point of evasive malware. Similar to our work, they also use sequence alignment, but only perform a coarse-grained alignment of system calls.

Static methods. Flake [9] proposed comparing the control flow graph of functions in two binaries for identifying similar code. This approach has been implemented in the industry-standard binary diffing tool BinDiff [10], which uses a heuristic graph-isomorphism algorithm for comparing CFGs of functions. Bourquin et al. [11] proposed an improved graph-isomorphism method for this problem. Since the structure of CFGs in a binary is often changed when using different compilers or optimization levels, the accuracy of CFG matching typically drops dramatically in such cases [23].

BinHunt [12] and iBinHunt [13] use symbolic execution to find semantically identical basic blocks in two binaries. Luo et al. [7] proposed using a theorem prover to find longest-common subsequences of semantically equivalent basic blocks for matching obfuscated code. Scalability is unfortunately a significant problem for approaches based on symbolic execution. Chandramohan et al. [14] proposed to ameliorate this problem by pre-filtering potential function matches before semantic comparison, and suggested *selective inlining* to cope with inlined library code. Pewny et al. [1] proposed semantic hashes using randomly sampled input/output pairs of basic blocks for finding known-vulnerable code across different architectures. David et al. [3] proposed a statistical framework for matching functions by aggregating semantic similarity scores of function code fragments. Eschweiler et al. [2] match functions using structural features of the CFG, and improve scalability by using pre-filtering based on simple numeric features. Feng et al. [4] pointed out that pre-filtering can lead to many false negatives, and instead proposed translating CFGs to numeric feature vectors to address scalability.

VII. Discussion and Future Work

Semantic differences. We showed in Section IV-B that our method is resilient to various semantics-preserving code transformations. While our case study indicated that the method can also cope with semantic changes, we intend to study this more rigorously in future work.

Evaluation on other architectures. Since our prototype implementation is based on Pin, it is currently limited to the x86 and x86-64 architectures. In future work, we intend to also evaluate our method's performance on other popular architectures, such as ARM or MIPS. Since the method is entirely architecture-agnostic, and due to its resilience to other code transformations (Section IV-B), we are hopeful that it will also allow cross-architecture alignments for other architectures. Due to our modular design, supporting other architectures only requires implementing a new trace recorder component.

Limitations of value-based analysis. We use runtime-observed results of computations as a proxy for the actual semantics of computations, and use DTW to align value traces. While this facilitates a scalable way to match long instruction traces, it has three major limitations. First, it requires both traces to be produced using the exact same program input and runtime environment. Second, it requires that all computations throughout both traces are carried out on the same concrete values. Finally, while our method can tolerate small local discrepancies in sequential consistency, global-scale reordering of computations cannot be handled by DTW.

The first and second limitations can be problematic if parts of a program's computations are nondeterministic. For example, many cryptographic protocols use random nonces, which makes recording two executions with identical inputs infeasible. One potential way to address this challenge is to directly compute the semantic similarity of code sections, rather than using observed runtime values. While scalability still remains an issue with current approaches, several recent works have proposed methods for semantic similarity comparisons of basic blocks [3], [14], [1], [7], [12], [13].

If only parts of a trace can be aligned in a meaningful way, e.g. due to the second or third limitation above, subsequence alignment can be used. This, however, requires the analyst to identify alignable parts of traces a priori. While visualization of the cost matrix can sometimes be used to identify such sections, exploring more rigorous approaches to solving this problem would be an important topic of future work. *Local sequence alignment* using the Smith-Waterman algorithm [25] can be used to find locally optimal subsequence alignments between two sequences, in cases where the full sequences cannot be globally aligned. Local sequence alignment is used extensively in bioinformatics for e.g. aligning DNA sequences. Since the Smith-Waterman algorithm is designed to align strings, its cost function is defined in terms of match or mismatch between letters. Adapting it to the problem of trace alignment would therefore require defining a similarity threshold for when two trace segments "match". Determining such a threshold is unfortunately difficult in the general case, but adapting local sequence alignment to our problem setting nonetheless remains an interesting direction for future work.

Concurrency may also pose a problem for trace alignment. Assuming that the computations in each thread are deterministic, our system can align traces of individual threads separately. However, in some cases, e.g. when using thread pools, the set of computations performed in each thread may vary. Our method can currently not handle such programs.

VIII. Conclusion

In this paper, we proposed a novel approach for aligning binary code traces, using dynamic time warping and techniques from information retrieval. We showed that our method is resilient to a number of code transformations, and can align code across 32 and 64-bit architectures. We also demonstrated that the method scales to traces with tens of millions of instructions or more. Finally, we also presented a practical use-case, showing how our method can aid in reverse engineering legacy binaries.

REFERENCES

[1] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 709–724.

[2] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "DiscovRE: Efficient cross-architecture identification of bugs in binary code," in *2016 Network and Distributed System Security Symposium*, 2016.

[3] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016, pp. 266–280.

[4] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[5] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 756–765.

[6] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 111–121.

[7] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 389–400.

[8] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 117–128.

[9] H. Flake, "Structural comparison of executable objects," *DIMVA 2004, July 6-7, Dortmund, Germany*, 2004.

[10] "Zynamics BinDiff," https://www.zynamics.com/bindiff.html.

[11] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 4.

[12] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[13] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 92–109.

[14] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 678–689.

[15] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *IEEE Symposium on Security and Privacy*, 2008, pp. 143–157.

[16] M. Müller, *Information retrieval for music and motion*. Springer, 2007.

[17] S. Salvador and P. Chan, "FastDTW:Toward Accurate Dynamic Time Warping in Linear Time and Space," *Intelligent Data Analysis*, vol. 11, no. 5, pp. 561–580, Oct. 2007.

[18] U. Kargén and N. Shahmehri, "Towards accurate binary correspondence using runtime-observed values," in *2016 IEEE International Conference on Software Maintenance and Evolution*, 2016, pp. 438–442.

[19] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2005, pp. 190–200.

[20] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, 2015, pp. 3–9.

[21] X. Zhang and R. Gupta, "Matching execution histories of program versions," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 197–206.

[22] V. Nagarajan, R. Gupta, X. Zhang, M. Madou, B. De Sutter, and K. De Bosschere, "Matching control flow of program versions," in *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 84–93.

[23] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 303–317.

[24] D. Kirat and G. Vigna, "Malgene: Automatic extraction of malware analysis evasion signature," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 769–780.

[25] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.