Tracelet-Based Code Search in Executables

Yaniv David  
Technion, Israel  
yanivd@cs.technion.ac.il

Eran Yahav  
Technion, Israel  
yahave@cs.technion.ac.il

Abstract

We address the problem of code search in executables. Given a function in binary form and a large code base, our goal is to statically find similar functions in the code base. Towards this end, we present a novel technique for computing similarity between functions. Our notion of similarity is based on decomposition of functions into tracelets: continuous, short, partial traces of an execution. To establish tracelet similarity in the face of low-level compiler transformations, we employ a simple rewriting engine. This engine uses constraint solving over alignment constraints and data dependencies to match registers and memory addresses between tracelets, bridging the gap between tracelets that are otherwise similar. We have implemented our approach and applied it to find matches in over a million binary functions. We compare tracelet matching to approaches based on n-grams and graphlets and show that tracelet matching obtains dramatically better precision and recall.

Categories and Subject Descriptors F.3.2(D.3.1)[Semantics of Programming Languages: Program analysis]; D.3.4 [Processors: compilers, code generation];

Keywords x86; x86-64; static binary analysis

1. Introduction

Every day hundreds of vulnerabilities are found in popular software libraries. Each vulnerable component puts any project that incorporates it at risk. The code of a single vulnerable function might have been stripped from the original library, patched, and statically linked, leaving a ticking time-bomb in an application but no effective way of identifying it.

We address this challenge by providing an effective means of searching within executables. Given a function in binary form and a large code base, our goal is to statically find similar functions in the code base. The main challenge is to define a notion of similarity that goes beyond direct syntactic matching and is able to find modified versions of the code rather than only exact matches.

Existing Techniques Existing techniques for finding matches in binary code are often built around syntactic or structural similarity only. For example, [15] works by counting mnemonics (opcode names, e.g., `mov` or `add`) in a sliding window over program text. This approach is very sensitive to the linear code layout, and produces poor results in practice (as shown in our experiments in Section 6). In fact, in Section 5, we show that the utilization of these approaches critically depends on the choice of a threshold parameter, and that there is no single choice for this parameter that yields reasonable precision/recall.

Techniques for finding exact and inexact clones in binaries employed n-grams, small linear snippets of assembly instructions, with normalization (linear naming of registers and memory locations) to address the variance in names across different binaries. For example, this technique was used by [20]. However, as instructions are added and deleted, the normalized names diverge and similarity becomes strongly dependent on the sequence of mnemonics (in a chosen window).

A recent approach [13] combined n-grams with graphlets, small non-isomorphic subgraphs of the control-flow graph, to allow for structural matching. This approach is sensitive to structural changes and does not work for small graphlet sizes, as the number of different (real) graphlet layouts is very small, leading to a high number of false positives.

In all existing techniques, accountability remains a challenge. When a match is reported, it is hard to understand the underlying reasons. For example, some matches may be the result of a certain mnemonic appearing enough times in the code, or of a block-layout frequently used by the compiler to implement a while loop or a switch statement (in fact, such frequent patterns can be used to identify the compiler [18, 19]).

Recently, [22] presented a technique for establishing equivalence between programs using traces observed during a dynamic execution. A static semantic-based approach was presented in [16], where abstract interpretation was used to establish equivalence of numerical programs. These techniques are geared towards checking equivalence and less suited for finding partial matches.

In contrast to these approaches, we present a notion of similarity that is based on tracelets, which capture semantic similarity of (short) execution sequences and allow matching similar functions even when they are not equivalent.

Tracelet-based matching Our approach is based on two key ideas:

- **Tracelet decomposition**: We use similarity by decomposition, breaking down the control-flow graph (CFG) of a function into tracelets: continuous, short, partial traces of an execution. We use tracelets of bounded length \(k\) (the number of basic blocks), which we refer to as \(k\)-tracelets. We bound the length of tracelets in accordance with the number of basic blocks, but a tracelet itself is comprised of individual instructions. The idea is that a tracelet begins and ends at control-flow instructions, and is otherwise a continuous trace of an execution. Intuitively, tracelet decomposition captures (partial) flow.

- **Tracelet similarity by rewriting**: To measure similarity between two tracelets, we define a set of simple rewrite rules and measure how many rewrites are required to reach from one tracelet to another. This is similar in spirit to recent approaches...
for automatic grading [23]. Rather than exhaustively searching the space of possible rewrite sequences, we encode the problem as a constraint-solving problem and measure distance using the number of constraints that have to be violated to reach a match. Tracelet rewriting captures distance between tracelets in terms of transformations that effectively “undo” low-level compiler decisions such as memory layout and register allocation. In a sense, some of our rewrite rules can be thought of “register deallocation” and “memory reallocation.”

**Main contributions:**

- A framework for searching in executables. Given a function in stripped binary form (without any debug information), and a large code base, our technique finds similar functions with high precision and recall.
- A new notion of similarity, based on decomposition of functions into *tracelets*: continuous, short, partial traces of an execution.
- A simple rewriting engine which allows us to establish tracelet similarity in the face of compiler transformations. Our rewriting engine works by solving alignment and data dependencies constraints to match registers and memory addresses between tracelets, bridging the gap between tracelets that are otherwise similar.

We have implemented our approach in a tool called TRACY and applied it to find matches in a code base of over a million stripped binary functions. We compare tracelet matching to other (aforementioned) approaches that use n-grams and graphlets, and show that tracelet matching obtains dramatically better precision and recall.

## 2. Overview

In this section, we give an informal overview of our approach.

### 2.1 Motivating Example

Consider the source code of Fig. 1(a) and the patched version of this code shown in Fig. 2(a). Note that both functions have the same format string vulnerability (\texttt{printf(optionalMsg)}, where \texttt{optionalMsg} is unsanitized and used as the format string argument). In our work, we would like to consider \texttt{doCommand1} and \texttt{doCommand2} as similar. Generally, this will allow us to use one vulnerable function to find other similar functions which might be vulnerable too. Despite the similarity at the source-code level, the assembly code produced for these two versions (compiled with gcc using default optimization level \texttt{O2}) is quite different. Fig. 1(b) shows the assembly code for \texttt{doCommand1} as a control-flow graph (CFG) \(G_1\), and Fig. 2(b) shows the code for \texttt{doCommand2} as a CFG \(G_2\). In these CFGs, we numbered the basic blocks for presentation purposes. We used the numbering \(n\) for a basic block in the original program, \(n'\) for a matching basic block in the patched program, and \(m^+\) for a new block in the patched program.

- Trying to establish similarity between these two functions at the binary level, we face (at least) two challenges: the code is structurally different and syntactically different. In particular:
  - The control flow graph structure is different. For example, block 6∗ in \(G_2\) does not have a corresponding block in \(G_1\).
  - The offsets of local variables on the stack are different, and in fact all variables have different offsets between versions. For example, \texttt{var\_18} in \(G_1\) becomes \texttt{var\_28} in \(G_2\) (this also holds for immediate values).
  - Different registers are used for the same operations. For example, block 3 of \(G_1\) uses \texttt{ecx} to pass a parameter to the \texttt{printf} call, while the corresponding block 3' in \(G_2\) uses \texttt{ebx}.

![Figure 1. \texttt{doCommand1} and its corresponding CFG \(G_1\).](image)

(iv) All of the jump addresses have changed, causing jump instructions to differ syntactically. For example, the target of the jump at the end of block 1 is \texttt{loc\_401358}, and in the corresponding jump in block 1', the jump address is \texttt{loc\_401370}.

Furthermore, the code shown in the figures assumes that the functions were compiled in the same context (surrounding code). Had the functions been compiled under different contexts, the generated assembly would differ even more (in any location-based symbol, address or offset).

**Why Tracelets?** A tracelet is a partial execution trace that represents a single partial flow through the program. Using tracelets has the following benefits:

- **Stability with respect to jump addresses:** Generally, jump addresses can vary dramatically between different versions of the same code (adding an operation can create an avalanche of address changes). Comparing tracelets allows us to side-step this problem, as a tracelet represents a single flow that implicitly and precisely captures the effect of the jump.

- **Stability to changes:** When a piece of code is patched locally, many of its basic blocks might change and affect the linear layout of the binary code. This makes approaches based on structural and syntactic matching extremely sensitive to changes. In contrast, under a local patch, many of the tracelets of the origi-
The idea of tracelets is natural from a semantic perspective. Taking tracelet length to be the maximal length of acyclic paths through a procedure is similar in spirit to path profiling [5, 17]. The problem of similarity between tracelets (loop-free sequences of assembly instructions) also arises in the context of super-optimization [6].

Figure 2. doCommand2 and its corresponding CFG $G_2$.

Figure 3. A pair of 3-tracelets: (i) based on the blocks (1,3,5) in Fig. 1(b), and (ii) on the blocks (1',3',5') in Fig. 2(b). Note that jump instructions are omitted from the tracelet, and that empty lines were added to the original tracelet to align similar instructions.

### 2.2 Similarity using Tracelets

Consider the CFGs $G_1$ and $G_2$ of Fig. 1 and Fig. 2. In the following, we denote 3-tracelets using triplets of block numbers. Note that in a tracelet all control-flow instructions (jumps) are removed (as we show in Fig. 3). Decomposing $G_1$ into 3-tracelets result in the following sequences of blocks:

$\ldots (1, 2, 4), (1, 2, 5), (1, 3, 5), (2, 4, 5), \ldots$

and doing the same for $G_2$ results in the following:

$\ldots (1', 2', 4'), (1', 2', 6'), (1', 3', 5'), (2', 4', 5'), (2', 6', 7'), (6', 7', 5') \ldots$

To establish similarity between the two functions, we measure similarity between the sets of their tracelets.

**Tracelet comparison as a rewriting problem**

As an example of tracelet comparison, consider the tracelets based on (1, 3, 5) in $G_1$, and (1', 3', 5') in $G_2$. In the following, we use the term *inSTRUCTION* to refer to an opcode (mnemonic) accompanied by its required operands. The two tracelets are shown in Fig. 3 with similar instructions aligned (blanks added for missing instructions). Note that in both tracelets, all control-flow instructions (jumps) have been omitted (shown as strikethrough). For example, in the original tracelet, *je short loc_401358* and *jmp short loc_40132F* were omitted as the flow of execution has already been determined.

A trivial observation about any pair of $k$-tracelets is that syntactic equality results in semantic equivalence and should be considered a perfect match, but even after alignment this is not true for these two tracelets. To tackle this problem, we need to be able to measure and grade the similarity of two $k$-tracelets that are not equal. Our approach is to measure the distance between two $k$-
tracelets by the number of rewrite operations required to edit one of them into the other (an edit distance). Note that a lower grade means greater similarity, where 0 is a perfect match. To do this we define the following rewrite rules:

- Delete an instruction \([\text{instDelete}]\).
- Add an instruction \([\text{instAdd}]\).
- Substitute one operand for another operand of the same type \([\text{Opr-for-Opr}]\) (the possible types are register, immediate and memory offset). Substituting a register with another register is one example of a replace rule.
- Substitute one operand for another operand of a different type \([\text{Opr-for-DiffOpr}]\).

Given a pair of \(k\)-tracelets, finding the best match using these rules is a rewriting problem requiring an exhaustive search solution. Following code patches or compilation in a different context, some syntactic deviations in the function are expected and we want our method to accommodate them. For example, a register used in an operation can be swapped for another depending on the compiler’s register allocation. We therefore redefine the cost in our rewriting rules such that if an operand is swapped with another operand throughout the tracelet, this would be counted at most once.

Fig. 4 shows an example of a score calculation performed during rewriting. In this example, the patched block on the left is rewritten into the original block on the right, using the aforementioned rewrite rules and assuming some cost function \(\text{Cost}\) used to calculate the cost for each rewrite operation. We present a way to approximate the results of this process, using 2 co-dependent heuristic steps: tracelet alignment and constraint-based rewriting.

**Tracelet alignment:** To compare tracelets we use a \(\text{tracelet alignment}\) algorithm, a variation on the longest common subsequence algorithm adapted for our purpose. The alignment algorithm matches similar operations and ignores operations added as a result of a patch. In our example, most of the original paths exist in the patched code (excluding \(1, 2, 5\), which was “lost” in the code change), and so most of the tracelets could also be found using patch-tolerant methods to compare basic blocks.

**Constraint-based matching:** To overcome differences in symbols (registers and memory locations), we use a tracelet from the original code as a “template” and try to rewrite a tracelet from the patched code to match it. This is done by addressing this rewrite problem as a constraint satisfaction problem (CSP), using the tracelets’ symbols as variables and its internal dataflow and the matched instruction alignment as constraints. Fig. 5 shows an example of the full process, alignment and rewriting, performed on the basic blocks 3 and 3’. It is important to note that this process is meant to be used on whole tracelets; we use only one basic block for readability. As we can see, the system correctly identifies the added instruction \((\text{mov esi, 4})\) and ignores it. In the next step, the patched tracelet’s symbols are abstracted according to the symbol type, and later successfully assigned to the correct value for a perfect match with the original tracelet. Note that function calls alone (or any other group of special symbols) could not have been used instead of this full matching process. For example, “printf” is very common in our code example, but it is only visible by name because it is imported; if it were an internal function, its name would have been stripped away.

### 3. Preliminaries

In this section, we provide basic definitions and background that we use throughout the paper.

**Assembly Instructions** An assembly instruction consists of a mnemonic, and up to 3 operands. Each operand can be one argument (register or immediate value) or a group of arguments used to address a certain memory offset. Fig. 6 provides a simplified grammar of x86 instructions. Note that in the case of OffsetCalc a group of arguments will be used to determine a single offset, and the value at this memory offset will serve as the operand for the instruction (see examples in the following table.)

We denote the set of all assembly instructions by \(\text{Instr}\). We define a set \(\text{Reg}\) of general purpose registers, allowing explicit read and write, a set \(\text{Imm}\) of all immediate values, and a set \(\text{Arg} = \text{Reg} \cup \text{Imm}\) containing both sets. Given an assembly instruction, we define the following:

- \(\text{read}(\text{Instr}) : \text{Instr} \rightarrow 2^{\text{Reg}}\), the set of registers being read by the instruction \(\text{Instr}\). A register \(r\) is considered as being read in an instruction \(\text{Instr}\) when it appears as the right-hand-side argument, or if it appears as a component in the computation of...
4. Tracelet-Based Matching

In this section, we describe the technical details of tracelet-based function similarity. In Section 4.1 we present a preprocessing phase applied before our tracelet-based similarity algorithm, the goal of this phase is to reverse some compilation side-effects. In Section 4.2 we present the main algorithm, an algorithm for comparing functions. The main algorithm uses tracelet similarity, the details of which are presented in Section 4.3, and a rewrite engine which is presented in Section 4.4.

4.1 Preprocessing: Dealing with Compilation Side-Effects

When the code is compiled into assembly, a lot of data is lost, or intentionally stripped, especially memory addresses.

Some data can be easily and safely recovered by abstracting any location-dependent argument values, while making sure that these changes will not introduce false matches. More generally, we perform the following substitutions (all examples are taken from Fig. 2(b)):

- Replace function offsets of imported functions with the function name. For example, `call 0x00401FF0` was replaced with `call _printf`.
- Replace each offset pointing to initialized global memory with a designated token denoting its content. For example, the address `0x00400002`, containing the string "DONE", was replaced with aCmdAddDone
- If the called function can be detected using the import table, replace variable offsets (stack or global memory) with the variable’s name, retrieved from the function’s documentation. For example, `var_24` should be replaced with `format` (this was not done in this case to demonstrate the common case in which the called function is not imported and as such its argument are not known).

This leaves us with two challenges:

1. Inter-procedural jumps (function calls): As our executables are stripped from function names and any other debug information, unless the function was imported (e.g., as `fprintf`), we have no way of identifying and marking identical functions in different executables. In Section 4.4 we show how our rewrite engine addresses this problem.

2. Intra-procedural jumps: The use of the real address will cause a mismatch (e.g., `loc_40132f` in Fig. 1(b) and `loc_401339` in Fig. 2(b) point to corresponding basic blocks). In Section 4.2.1, we show how to extract tracelets that follow intra-procedural jumps.

4.2 Comparing Functions in Binary Code

Using the CFG to measure similarity When comparing functions in binary form, it is important to realize that the linear layout of the binary code is arbitrary and as such provides a poor basis for comparison. In the simple case of an if-then-else statement (splitting execution for the true and false clauses), the choice made by the compiler as to which clause will come first in the binary is determined by parameters such as basic block size or the likelihood of a block being used. Another common example is the layout of switch-case statements, which, aside from affecting a larger portion of the code, also allows for multiple code structures to be employed: a balanced tree structure can be used to perform a binary search on the cases, or a different structure such as a jump (lookup) table can be used instead.

To obtain a reasonable similarity metric, one must, at least, use the function’s control flow (alternatively, one can follow data
and control dependencies as captured in a program dependence graph \cite{10}). Following this notion, we find it natural to define the “function similarity” between two functions by extracting tracelets from each function’s CFGs and measuring the coverage rate of the matching tracelets. This coincides with our goal to create an accountable measure of similarity for assembly functions, in which the matched tracelets retrieved during the process will be clearly presented, allowing further analysis or verification.

**Function Similarity by Decomposition** Algorithm 1 provides the high-level structure of our comparison algorithm. We first describe this algorithm at a high-level and then present the details for each of its steps.

Algorithm 1: Similarity score between two functions

<table>
<thead>
<tr>
<th>Input:</th>
<th>TR - target and reference functions, NM - normalizing method: ratio or containment k - tracelet size, α, β - threshold values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>IsMatch - true if the functions are a match, SimilarityScore - the normalized similarity score</td>
</tr>
</tbody>
</table>

```
Algorithm FunctionsMatchScore (T,R,NM,k,α,β)
RefTracelets = ExtractTracelets(R,k)
TargetTracelets = ExtractTracelets(T,k)
MatchCount = 0
foreach r ∈ RefTracelets do
  foreach t ∈ TargetTracelets do
    AlignedInsts = AlignTracelets(r,t)
    t’ = RewriteTracelet(AssignedInsts,r,t)
    S = CalcScore(r,t')
    RId = CalcScore(r,r')
    TId = CalcScore(t,t')
    if Norm(S,RId,TId,NM) > β then
      MatchCount++
  end
isMatch = SimilarityScore > α
end
```

The algorithm starts by decomposing each of the two functions into a set of \( k \)-tracelets (lines 2-3). We define the extract notion of a tracelet later in this section; for now it suffices to view each tracelet as a bounded sequence of instructions without intermediate jumps. The `extractTracelets` operation is explained in Sec. 4.2.1. After each function has been decomposed into a set of tracelets, the algorithm proceeds by pairwise comparison of tracelets. This is done by first aligning each pair (Line 7, see Sec. 4.3), and then trying to rewrite the target tracelet using the reference tracelet (Line 8, see Sec. 4.4). We will later see that `CalcScore` and `AlignTracelets` actually perform the same operation and are separated for readability (see Sec. 4.3). The tracelet similarity score is calculated using the identity similarity scores (the similarity score of a tracelet with itself) for the target and reference tracelets (computed in lines 10-11), and one of two normalization methods (ratio or containment, Sec. 4.3). Two tracelets are considered similar, or a “match” for each other, if their similarity score is above threshold \( β \). After all tracelets were processed, the resulting number of similar tracelets is used to calculate the coverage rate, which acts as the similarity score for the two functions. Finally, two functions are considered similar if their similarity score is above the threshold \( α \).

### 4.2.1 Extracting tracelets

Following this, we formally define our working unit, the \( k \)-tracelet, and show how \( k \)-tracelets are extracted from the CFG. A \( k \)-tracelet is an ordered tuple of \( k \) sequences, each representing one of the basic blocks in a directed acyclic sub-path in the CFG, and containing all of the basic block’s assembly instructions excluding the jump instruction. Note that as all control-flow information is stripped, the out-degree of the last basic block (or any other basic block) in the tracelet is not used in the comparison process. This choice was made to allow greater robustness with regard to code changes and optimization side-effects, as exit nodes can be added and removed. Also note that the same \( k \) basic blocks can appear in a different order (in a different tracelet), if such sub-paths exist in the CFG. We only use small values of \( k \) (1 to 4), and because most basic blocks have 1 or 2 out-edges, and a few of which are back-edges, such duplicate tracelets (up to order) are not very common. Algorithm 2 shows the algorithm for extracting \( k \)-tracelets from a CFG. The tracelets are extracted recursively from the nodes in the CFG. To extract all the \( k \)-tracelets from a certain node in the CFG, we compute all \((k-1)\)-tracelets from any of its "sons", and use a Cartesian product \( \times \) between the node and the collected tracelets.

Algorithm 2: Extract all \( k \)-tracelets from a CFG

<table>
<thead>
<tr>
<th>Input:</th>
<th>G - control flow graph, k - tracelet size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>T - a list of all the tracelets in the CFG</td>
</tr>
</tbody>
</table>

```
Algorithm ExtractTracelets (G,k)
result = ∅
foreach b ∈ B do
  result = result ∪ Extract(b,k)
end
```

```
Function Extract (b,k)
  bCode = {stripJumps(b)}
  if k = 1 then
    return bCode
  else
    return \( \bigcup \{ \{ b' \} \in E \} \) bCode \( \times \) Extract(b',k-1)
end
```

### 4.3 Calculating Tracelet Match Score

**Aligning tracelets with an LCS variation** As assembly instructions can be viewed as a text string, one way to compare them is using a textual diff. This simulates the way a human might detect similarities between tracelets. Comparing them in this way might work to some extent. However, a textual diff might decompose an assembly instruction and match each decomposed part to a different instruction. This may cause very undesirable matches such as “\texttt{rorx edx,esi}” with “\texttt{inc rdi}” (shown in bold, \texttt{rorx} is a rotate instruction, and \texttt{rdi} is a 64-bit register).

A common variation of LCS is the edit distance calculation algorithm, which allows additions and deletions by using dynamic programming and a “match value table” to determine the “best” match between two strings. The match value table is required, for example, when replacing a space because doing so will be costlier than other replacement operations due to the introduction of a new “word” into the sentence. By declaring one of the strings as a reference and the other as the target, the output of the algorithm can be extended to include matched, deleted, and inserted letters. The matched letters will be consecutive substrings from each string, whereas the deleted and inserted letters must be deleted from or

\texttt{rdi}
inserted to the target, in order to match it to the reference string. A thorough explanation about edit-distance is presented in [25].

We present a specially crafted variation on the edit distance. We will treat each instruction as a letter (e.g., pop eax; jmp ebx; add esp, ebx; with itself is 3, whereas the score of add esp, eax; with add esp, ebx; is only 2. We compute the similarity between assembly instructions as follows:

\[
Sim(c, c') = \begin{cases} 
2 + |[i|\text{args}(c)[i] = \text{args}(c')[i]| & \text{SameKind}(c, c') \\
-1 & \text{otherwise.}
\end{cases}
\]

Here, c and c’ are the assembly instructions we would like to compare. As defined in Section 3, we say that both instructions (our letters) are the same “kind” if their mnemonic is the same, and all of their argument are of the same kind (this will prove even more valuable in our rewrite process). We then calculate the grade by counting the number of matching arguments, also giving 2 “points” for the fact that the “kinds” match. If the instructions are not the same kind, the match is given a negative score and won’t be favored for selection. This process enables us to accommodate code changes; for example, we can “jump over” instructions which were heavily changed or added, and match the instructions that stayed mostly similar but one register was substituted for another. Note that this metric gives a high value to instructions with a lot of arguments in common, and that some of these arguments were preprocessed to allow for better matches (Section 4.1), while others, such as registers, are architecture dependent.

Algorithm 3: Calculate similarity score between two tracelets

\[
\text{Input: } T, R - \text{target and reference tracelets} \\
\text{Output: } \text{Score} - \text{tracelet similarity score}
\]

\[
\begin{array}{l}
\text{Algorithm CalcScore}(T, R) \\
A = \text{InitMatrix}([T], [R]) \\
\text{// Access outside the array returns a large negative value} \\
\text{for } i = |T|; i > 0; i -- \text{ do} \\
\text{for } j = |R|; j > 0; j -- \text{ do} \\
A[i, j] = \text{Max} \\
\text{Sim}(T[i], R[j]) + A[i + 1, j + 1], \text{// match} \\
A[i + 1, j], \text{// insert} \\
A[i, j + 1], \text{// delete} \\
\end{array}
\]

\[
\text{Score} = A[0, 0]
\]

Using LCS based alignment Given a reference and a target tracelet, an algorithm for calculating the similarity score for the two tracelets is described in Algorithm 3. This specialized edit distance calculation process results in:

1. A set of aligned instruction pairs, one instruction from the reference tracelet and one from the target tracelet.
2. The similarity score for the two tracelets, which is the sum of \(Sim(c, c')\) for every aligned instruction pair c and c’.
3. A list of deleted and inserted instructions.

Note that the first two outputs were denoted AlignTracelets and CalcScore, respectively, in Algorithm 1.

Despite not being used directly in the algorithm, inserted and deleted data (for a “successful” match) give us important information. Examining the inserted instructions will uncover changes made from the reference to the target, such as new variables or changes to data structures. On the other hand, deleted instructions show us what was removed from the target, such as support for legacy protocols. Identifying these changes might prove very valuable for a human researcher, assisting with the identification of global data structure changes, or with determining that an underlying library was changed from one provider to another.

Normalizing tracelet similarity scores We used two different ways to normalize the tracelet similarity score (“Norm” in Algorithm 1):

1. Containment: requiring that one of the tracelets be contained in the other. Calculation: \(S/\min(R|\text{Ident}, T|\text{Ident})\).
2. Ratio: taking into consideration the proportional size of the unmatched instructions in both tracelets. Calculation: \((S + 2)/(R|\text{Ident} + T|\text{Ident})\).

Note that in both methods the normalized score is calculated using the similarity score for the reference and target tracelets, and the identity scores for the two tracelets.

Each method is better suited for different scenarios, some of which are discussed in Section 8, but, as our experiments show, for the common cases they provide the same accuracy.

Next we present a supplementary step that improves our matching process even further by performing argument “de-allocation” and rewrite.

4.4 Using the Rewrite Engine to Bridge the Gap

We first provide an intuitive description of the rewrite process, and then follow with a full algorithm accompanied by explanations.

Using aligned tracelets for argument de-allocation To further improve our matching process, and in particular to tackle the ripple effects of compiler optimizations, we employ an “argument de-allocation” and rewrite technique.

First, we abstract away each argument of the target tracelet to an unknown variable. These variables are divided into three groups: registers, memory locations, and function names. Next, we introduce the constraints representing the relations between the variables, using data flow analysis, and the relations between the variables and the matched arguments. Arguments are matched using their linear position in matched instructions. Because reference and target tracelets were extracted from real code, we will use them as “hints” for the assignment by generating two sets of constraints: in-tracelet constraints and cross-tracelet constraints. The in-tracelet constraints preserve the relation between the data and arguments inside the target tracelet. The second stage will introduce the cross-tracelet constraints, inducing argument equality for every argument in the aligned instructions. Finally, combining these constraints and attempting to solve them by finding an assignment with minimum conflicts is the equivalent of attempting to rewrite the target tracelet into the reference tracelet, while respecting the target’s data flow and memory layout. Note that if all constraints are broken, the assignment is useless as the score will not improve, but this means that the tracelets are not similar and reducing their similarity score improves our method’s precision.

Returning to the example of Section 2, the full process of the tracelet alignment and rewrite is shown in Fig. 5. As we can see, in this example the target tracelet achieved a near perfect score for the new match (only “losing points” for the missing instruction if ratio calculation was used).

Algorithm details Our rewriting method is shown in Algorithm 4. First, we iterate over the aligned instruction pairs from T and R. For every pair of instructions, \(|\text{arg}(r)| \neq |\text{arg}(t)|\) holds (this is required for aligned instructions; see Section 4.3). For every pair of aligned arguments (aligned with regard to their linear position, e.g., eax ↔ ebx, ecx ↔ edx, in add eax, ecx; add ebx, edx),
we abstract the argument in the target tracelet. This is done using newVarName, which generates unique temporary variable names according to the type of the argument (such as the ones shown in Fig. 5). Then, a cross-tracelet constraint between the new variable and the value of the argument from the reference instruction is create and added to the constraint \( \varphi \) (line 7).

Then, read\( (t) \) is used to determine whether the argument \( s_t \) is read in \( t \). Next, the algorithm uses the helper data structure lastWrite\( (s_t) \) to determine the last temporary variable name in which the argument \( s_t \) was written into. It then creates a dataflow constraint from the last write to the read (line 9). Otherwise the algorithm checks whether \( t \) is a write instruction on \( s_t \), using write\( (t) \), and updates lastWrite\( (s_t) \) accordingly.

Finally, after creating all variables and constraints, we call the constraint solver (using solve, line 14), obtain a minimal conflict solution, and use it to rewrite the target tracelet.

\begin{algorithm}
\caption{Rewrite target code to match reference.}
\begin{algorithmic}[1]
\State \textbf{Input:} \texttt{AlignedInstrs} - a set of tuples with aligned reference and target assembly instructions (Note that aligned instructions have the same number of arguments), \texttt{T,R} - target and reference tracelets
\State \textbf{Output:} \texttt{T'} - rewritten target code
\Function{RewriteTracelet} {\texttt{T,R}}
\For {\texttt{(t,r) \in AlignedInstrs}}
\State \texttt{s_t = args(t)[i]; i + + do}
\State \texttt{s_r = args(r)[i]}
\State \texttt{nv = newVarName(s_t)}
\State \texttt{\varphi = \varphi \land (nv = s_r)}
\If {\texttt{s_t \in read(t) and lastWrite(s_t) \neq \bot}}
\State \texttt{\varphi = \varphi \land (nv = lastWrite(s_t))}
\ElseIf {\texttt{s_t \in write(t) then}}
\State \texttt{lastWrite(s_t) = nv}
\EndIf
\EndFor
\State \texttt{vmap = solve(\varphi, symbols(R))}
\For {\texttt{t \in T'}}
\State \texttt{t' = t}
\For {\texttt{s_t \in t do}}
\If {\texttt{s_t \in vmap then}}
\State \texttt{t' = t'[s_t \mapsto vmap(s_t)]}
\EndIf
\EndFor
\State T'.append(t')
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

**Constraint solver details** In our representation, we identify registers and allow them to be replaced by other registers. Our domain for the register assignment only contains registers found in the reference tracelet. Generally, two operations that involve memory access may be considered similar when they perform the same access to the same variable. However, the problem of identifying variables in striped binaries is known to be extremely challenging [4]. We therefore choose to consider operations as similar under the more strict condition that all the components involved in the computation of the memory address are syntactically identical. Our rewrite engine therefore adds constraints that try to match the components of memory-address computation across tracelets.

The domains for the global offsets in memory and function call arguments again contain exactly the arguments of the reference tracelet. Note that each constraint is a conjunction (lines 7,9), and

Figure 7. A’s members are dissimilar to B’s. We will measure precision and recall on R=A.

5. Evaluation

5.1 Test-bed Structure

**Test-bed design** We built our testbed to cover all of the previously discussed cases in which we attempt to detect similarity: when a code is compiled in a different context, or after a patch. Accordingly, we divided the executables into two groups:

- Context group: This group is comprised of several executables containing the same library function. We used Coreutils executables and focused on library functions which perform the common task of parsing command line parameters.

- Code Change group: This group includes functions originating from several executables which are different versions of the same application. For example, we used \texttt{wget} versions 1.10.1.12.1.14.

In the first step of our evaluation process we performed controlled tests: we built all the samples from the source code, used the same machines, and the same default configurations. This allowed us to perform extensive testing using different test configurations (i.e., using different values of \( k \) and with different values of \( \beta \)). During this stage, as we had access to the source code, we could perform tests on edge conditions, such as deliberately switching the similar functions between the executables (“functions implants”) and measuring code change at the source level to make sure the changes are reasonable. We should note that when building the initial test set, the only requirement from our functions was that they be “big” enough, that is, that they have more than 100 basic blocks. This
CROC is a recent improvement of ROC that addresses the problem of “early retrieval,” where there is a huge number of potential matches and the number of real matches is known to be very low. The CROC metric is described in detail in [24]; the idea behind it is to better measure the accuracy of a low number of matches. This is appropriate in our setting because manually verifying that a match is real is a very costly operation for a human expert. Moreover, software development is inherently based on re-use, and similar functions will not naturally appear in the same executable (so each executable will contain at most one true positive). CROC gives a stronger grade to methods that provide a low number of candidate matches for a query (i.e., it penalizes false positives more aggressively than ROC).

5.2 Prototype Implementation

We have implemented a prototype of our approach in a tool called TRACY. In addition to tracelet-based comparison, we have also implemented the other aforementioned comparison methods, based on a sliding window of mnemonics (n-grams) and graphlets. This was done in an attempt to test the different techniques on the same data, measuring precision and recall (and ignoring performance, as our implementation of the other techniques is non-optimized).

Our prototype obtains a large number of executables from the web and stores them in a database. A tracelet extraction component then disassembles and extracts tracelets from each function in the executables stored in the database, creating a search index. Finally, a search engine allows us to use different methods to compare a query (a function given in binary form, as a part of an executable) to the disassembled binaries in the index.

The system itself was implemented almost entirely in Python using IDA Pro [2] for disassembly, iGraph to process graph, and MongoDB for storing and indexing, and yard-plot ([3]) for plotting ROC and CROC curves. The full source code can be found at https://github.com/YanivMD/Tracy.

The prototype was deployed on a server with four quad-core Intel Xeon CPU E5-2670 (2.60GHz) processors, and 188 GiB of RAM, running Ubuntu 12.04.2 LTS.

Optimizations and parallel execution As shown in Table 1, the number of tracelet compare operations is huge, but as similarity score calculations on pairs of tracelets are independent, they can be done in parallel. One important optimization is first performing the instruction alignment (using our LCS algorithm) with respect to the basic-block boundary. This reduces the calculation’s granularity and allows for even more parallelism. Namely, when aligning two traces, $\text{(1, 2, 3)}$ and $(1', 2', 3')$, instructions from 1 can only be matched and aligned with instructions from $1'$. This allowed us to use 1-tracelets (namely basic blocks) to cache scores and matches, and then use these alignments in the k-tracelet’s rewriting process. This optimization doesn’t affect precision because the rewrite process uses all of the k nodes in the tracelet, followed by a full re-calculation of the score (which doesn’t use any previous data). Furthermore, to better adapt our method to work on our server’s NUMA architecture, we statically split the tracelets of the representative function (which is compared with the rest of the functions in the database) across the server’s nodes. This allowed for better use of the core’s caches and avoided memory overheads.

5.3 Test Configuration

Similarity calculation configuration The following parameters require configuration.

- $k$, the number of nodes in a tracelet.
- The “tracelet match barrier” ($\beta$ in Algo. 1), a threshold parameter above which a tracelet will count as a match for another.
6. Results

6.1 Comparing Different Configurations

Testing different thresholds for tracelet-to-tracelet matches Table 2 shows the results of using different thresholds for 3-tracelet to 3-tracelet matches. Higher border values (excluding 100) lead to better accuracy. This makes sense as requiring a higher similarity score between the tracelets means that we only match similar tracelets. All border values between 70 and 90 percent give the same accuracy, suggesting that similar tracelets score above 90, and that dissimilar tracelets score below 70. Our method thus allows for a big “safety buffer” between the similar and dissimilar spectrums. It is, therefore, very robust. Finally, we see that for 100 (requiring a perfect syntactical match), we get a lower grade. This makes sense as we set out to compare “similar” tracelets knowing (even after the rewrite) that they still contain code changes which cannot (and should not) be matched.

Testing different values of $k$ The last parameter in our tests is $k$ (the number of basic blocks in a tracelet). For each value of $k$, we attempted every threshold for tracelet-to-tracelet matching (as done in the previous paragraph). Using $k = 1$, we achieved a relatively low CROC AUC of 0.83. Trying instead $k = 2$ yielded 0.91, while $k = 3$ yielded 0.99. The results for $k = 4$ are similar to the results of $k = 3$ and are only slightly more expensive to compute (the number of 4-tracelets is similar to that of 3-tracelets; see Table 1). In summary, using a higher number for $k$ is crucial for accuracy. The less accurate results for lower $k$ values is due to their leading to shorter tracelets having fewer instructions to match and fewer constraints (especially in-tracelet), resulting in lower precision.

Using ratio and containment calculations Our experiments show that this parameter does not affect accuracy. This might be because containment or ratio are traits of whole functions and so are marginal in the scope of the tracelet.

Detecting vulnerable functions After our system’s configuration was calibrated, we set out to employ it for its main purpose, finding vulnerable executables. To make this test interesting, we looked for vulnerabilities in libraries because such vulnerabilities affect multiple applications. One example is CVE-2010-0624 ([1]); this vulnerability is an exploitable heap-based buffer overflow affecting GNU tar (up to 1.22) and GNU cpio (up to 2.10). We compiled a vulnerable version of tar on our machine and scanned package repositories, looking for vulnerable versions of cpio and tar. This resulted in our system successfully pinpointing the vulnerable function in the tar 1.22 packages, the older tar 1.21 packages, and packages of cpio 2.10. (older packages of cpio and tar were not found at all and so could not be tested).

<table>
<thead>
<tr>
<th>$\beta$ Value</th>
<th>10 - 20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70 - 90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUC[CROC]</td>
<td>0.15</td>
<td>0.23</td>
<td>0.45</td>
<td>0.78</td>
<td>0.95</td>
<td>0.99</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 2. Showing the CROC AUC score for the tracelet-based matching process using different values of $\beta$

- The "function coverage rate match barrier" ($\alpha$ in Algo. 1), a threshold parameter, above which a function will be considered similar to another one.

Because the ROC (and CROC) methods attempt all values for $\alpha$, for every reasonable value of $k$ ($1 \leq k \leq 4$), we ran 10 experiments testing all values of $\beta$ (We checked values from 10 to 100 percent match, in 10 percent intervals) to discover the best values to use. The best results for each parameter is reported in the next section.

For our implementation of the other methods, we used the best parameters as reported in [13], using n-gram windows of 5 instructions with a 1 instruction delta, and $k=5$ for graphlets.

6.2 Using the Rewrite Engine to Improve Tracelet Match Rate

Fig. 8 shows executables containing a function which was successfully classified as similar to quotearg_buffer_restyled (coreutils library), compiled locally inside $wc$ during the first stage of our experiments. Following is a breakdown of the functions detected:

- The exact function compiled locally into different executables (in the controlled test stage), such as ls, rm and chmod.
- The exact function “implanted” in another version of the executable. For example, $wc$ from 7.6 means that the version of quotearg_buffer_restyled from the coreutils 7.6 library was implanted in another version (in this case 8.19).

|                | n-grams | Graphlets $K=5$ | Tracelets $K=3$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AUC[CROC]</td>
<td>0.7217</td>
<td>0.5913</td>
<td>1</td>
</tr>
<tr>
<td>AUC[CROC]</td>
<td>0.2451</td>
<td>0.1218</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 3. Accuracy for tracelet-based function similarity vs graphlets and n-grams, using ROC and CROC. These results are based on 6 different experiments.
• Different versions of the function found in executables downloaded from internet repositories, such as `deb wc 8.5` (the `wc` executables, Debian repository, coreutils v8.5).

To present the advantages of the rewrite engine, we separated the percentage of tracelets that were matched before the rewrite from the ones that could only be matched after the rewrite. Note that both match scores are obtained using our LCS derived match scoring algorithm run once before the rewrite, and once after. An average of 25% of the total number of tracelets were matched as a result of the rewriting process.

It should be noted that true positive and negative information was gathered using manual similarity checks and/or using debugging information (comparing function names, as we assume that if a function changed its name, it will probably have changed its purpose, since we are dealing with well-maintained code).

### 6.3 Performance

Our proof-of-concept implementation is written in Python and does not employ optimizations (not even caching, clustering, or other common search-engine optimization techniques). As a result, the following reported running times should only be used to get a grasp on the amount of work performed in each step of the process, show an upper limit, and achieve a better understanding of where optimizations should be employed. Analyzing complete matching operations (which may take from milliseconds to a couple of hours) shows that the fundamental operation in our process, that of comparing two tracelets, greatly depends on the composition of the tracelets being compared. If the tracelets are an exact match or are not remotely similar, the process is very fast because, during alignment, we do not need to check multiple options and the rewrite engine (which relies on the alignment and as such cannot be measured independently) does not need to do too much work. When the tracelets are similar but are not a perfect match, the rewrite engine has a chance to really assist in the matching process and as such requires some runtime.

Table 4 summarizes tracelet comparison, with and without the rewrite operation, showing that almost half of the time is spent in the rewrite engine (on average). Also presented in the table is the runtime of a complete function-to-function comparison. It should be noted that the run times greatly depend on the size of the functions being compared, and the shown run times were measured on functions containing ~ 200 basic-blocks. Postmortem analysis of the results of the experiments described earlier shows that tracelets with a matching score below 50%, which comprised 85% of the cases, will not be improved using the rewrite engine, and so a simple yet effective optimization will be to skip the rewriting attempts in such cases.

### 7. Related Work

We have discussed closely related work throughout the paper. In this section, we briefly survey additional related work.

**n-grams based methods** The authors of [12] propose a method for determining even complex lineage for executables. Nonetheless, at its core their method uses linear n-grams combined with normalization steps (in this case also normalizing jumps), which, as we discussed in Sec. 4.2, is inherently flawed due to reliance on the compiler to make the same layout choices.

**Structural methods** A method for comparing binary control flow graphs using graphlet-coloring was presented in [14]. This approach has been used to identify malware, and for identifying compilers and even authors [19],[18]. This method was designed to match whole binaries in large groups, and as such it employs a coarse equivalence criterion between graphlets. [8] is another interesting work in this field; it attempts to detect virus infections inside executables. This is done by randomly choosing a starting point in the executable, parsing it as code and attempting to construct an arbitrary length CFG from there. This work is focused on cleverly detecting the virus entry point, and presents interesting ideas for analyzing binary code that we can use in the future.

**Similarity measures developed for code synthesis testing** The authors of [21] propose an interesting way to compare x86 loop-free snippets to perform transformation correctness tests. The similarity is calculated by running the code on selected inputs, and quantifying similarity by comparing outputs and states of the machine at the end of the execution (for example counting equal bits in the output). This distance metric does offer a way to compare two code fragments and possibly to compute similarity, but requires dynamic execution on multiple inputs, which makes it infeasible for our cause.

**Similarity measures for source code** There has been a lot of work on detecting similarities in source code (cf. [7]). As our problem deals with binary executables, such approaches are inapplicable. (We discussed an adaptation of these approaches to binaries [20] in Sec. 1). Using program dependence graphs was shown effective in comparing functions using their source code [11], but applying this approach to assembly code is difficult. Assembly code has no type information, variables are not easily identified [4], and attempting to create a PDG proves costly and imprecise.

**Dynamic methods** There are several dynamic methods targeting malware. For example, [9] uses run-time information to model
We decompose programs into tracelets: continuous, short, partial traces of an execution. To efficiently compare tracelets (an operation that has to be applied frequently during search), we encode their matching as a constraint solving problem. The constraints capture alignment constraints and data dependencies to match registers and memory addresses between tracelets. We implemented our approach and applied it to find matches in over a million binary functions. We compared tracelet matching to approaches based on n-grams and graphlets and show that tracelet matching obtains dramatically better results in terms of precision and recall.

Acknowledgement

We would like to thank David A Padua, Omer Katz, Gilad Maymon, Nimrod Partush and the anonymous referees for their feedback and suggestions on this work, and Mor Shoham for his help with the system’s implementation. The research leading to these results has received funding from the European Union’s, Seventh Framework Programme (FP7) under grant agreement no. 615688.

References

[10] HORWITZ, S. Identifying the semantic and textual differences between two versions of a program. In PLDI ’90.
[18] ROSENBLUM, N., ZHU, X., AND MILLER, B. P. Who wrote this code? identifying the authors of program binaries. In ESORICS’11.