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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '14, June 9 - 11 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594343

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 (`printf(optionalMsg)`, where optionalMsg is unsanitized and used as the format string argument). In our work, we would like to consider `docommand1` and `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 `O2`) is quite different. Fig. 1(b) shows the assembly code for `docommand1` as a control-flow graph (CFG) G_1, and Fig. 2(b) shows the code for `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:

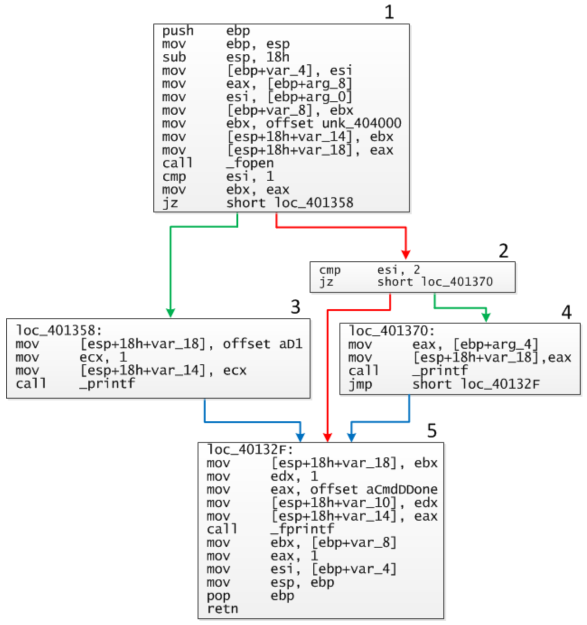
(i) The control flow graph structure is different. For example, block 6^* in G_2 does not have a corresponding block in G_1.

(ii) The offsets of local variables on the stack are different, and in fact *all* variables have different offsets between versions. For example, `var_18` in G_1 becomes `var_28` in G_2 (this also holds for immediate values).

(iii) Different registers are used for the same operations. For example, block 3 of G_1 uses `ecx` to pass a parameter to the `printf` call, while the corresponding block $3'$ in G_2 uses `ebx`.

```
int doCommand1(int cmd,char * optionalMsg,
        char * logPath) {
    int counter =1;
    FILE *f = fopen(logPath,"w");
    if (cmd == 1) {
        printf("(%d) HELLO",counter);
    } else if (cmd == 2) {
        printf(optionalMsg);
    }
    fprintf(f,"Cmd %d DONE",counter);
    return counter;
}
```

(a)



```
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+var_4], esi
mov     eax, [ebp+arg_8]
mov     esi, [ebp+arg_0]
mov     [ebp+var_8], ebx
mov     ebx, offset unk_404000
mov     [esp+18h+var_14], ebx
mov     [esp+18h+var_18], eax
call    _fopen
cmp     esi, 1
mov     ebx, eax
jz      short loc_401358
```

```
cmp     esi, 2
jz      short loc_401370
```

```
loc_401358:
mov     [esp+18h+var_18], offset aD1
mov     ecx, 1
mov     [esp+18h+var_14], ecx
call    _printf
```

```
loc_401370:
mov     eax, [ebp+arg_4]
mov     [esp+18h+var_18],eax
call    _printf
jmp     short loc_40132F
```

```
loc_40132F:
mov     [esp+18h+var_18], ebx
mov     edx, 1
mov     eax, offset aCmdDDone
mov     [esp+18h+var_10], edx
mov     [esp+18h+var_14], eax
call    _fprintf
mov     ebx, [ebp+var_8]
mov     eax, 1
mov     esi, [ebp+var_4]
mov     esp, ebp
pop     ebp
retn
```

(b) G_1

Figure 1. `doCommand1` and its corresponding CFG G_1.

(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 `loc_401358`, and in the corresponding jump in block 1', the jump address is `loc_401370`.

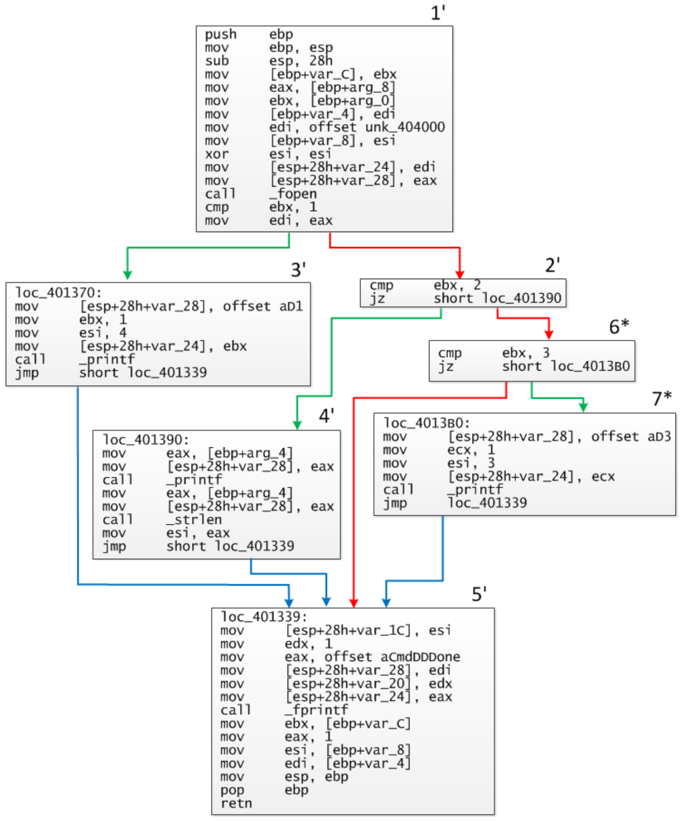
Furthermore, the code shown in the figures assumes that the functions were compiled in the same context (surrounding code). Had the functions been compiled under different contexts, the generated assembly would differ even more (in any location-based symbol, address or offset).

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-

```
int doCommand2(int cmd,char *optionalMsg,char *logPath){
    int counter = 1; int bytes = 0; // New variable
    FILE *f = fopen(logPath,"w");
    if (cmd == 1) {
        printf("(%d) HELLO",counter); bytes += 4;
    } else if (cmd == 2) {
        printf(optionalMsg); bytes+= strlen(optionalMsg);
    /* This option is new: */
    } else if (cmd == 3) {
        printf("(%d) BYE",counter); bytes += 3;
    }
    fprintf(f,"Cmd %d\\%d DONE",counter,bytes);
    return counter;
}
```

(a)



```
push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_C], ebx
mov     eax, [ebp+arg_8]
mov     ebx, [ebp+arg_0]
mov     [ebp+var_4], edi
mov     edi, offset unk_404000
mov     [ebp+var_8], esi
xor     esi, esi
mov     [esp+28h+var_24], edi
mov     [esp+28h+var_28], eax
call    _fopen
cmp     ebx, 1
mov     edi, eax
```

```
cmp     ebx, 2
jz      short loc_401390
```

```
loc_401370:
mov     [esp+28h+var_28], offset aD1
mov     ebx, 1
mov     esi, 4
mov     [esp+28h+var_24], ebx
call    _printf
jmp     short loc_401339
```

```
cmp     ebx, 3
jz      short loc_4013B0
```

```
loc_401390:
mov     eax, [ebp+arg_4]
mov     [esp+28h+var_28], eax
call    _printf
mov     eax, [ebp+arg_4]
mov     [esp+28h+var_28], eax
call    _strlen
mov     esi, eax
jmp     short loc_401339
```

```
loc_4013B0:
mov     [esp+28h+var_28], offset aD3
mov     ecx, 1
mov     esi, 3
mov     [esp+28h+var_24], ecx
call    _printf
jmp     loc_401339
```

```
loc_401339:
mov     [esp+28h+var_1C], esi
mov     edx, 1
mov     eax, offset aCmdDDDone
mov     [esp+28h+var_28], edi
mov     [esp+28h+var_20], edx
mov     [esp+28h+var_24], eax
call    _fprintf
mov     ebx, [ebp+var_C]
mov     eax, 1
mov     esi, [ebp+var_8]
mov     edi, [ebp+var_4]
mov     esp, ebp
pop     ebp
retn
```

(b) G_2

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

nal code remain similar. Furthermore, tracelets allow for alignment (also considering insertions and deletions of instructions), as explained in Sec. 4.3.

- **Semantic comparison:** Since a tracelet is a partial execution trace, it is feasible to check semantic equivalence between tracelets (e.g., using a SAT/SMT solver as in [21]). In general, this can be very expensive and we present a practical approach for checking equivalence that is based on data-dependence analysis and rewriting (see Sec. 4.4).

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].

```
push ebp
mov ebp, esp
sub esp, 18h
mov [ebp+var_4], esi
mov eax, [ebp+arg_8]
mov esi, [ebp+arg_0]
mov [ebp+var_8], ebx
mov ebx, offset unk_404000


mov [esp+18h+var_14], ebx
mov [esp+18h+var_18], eax
call _fopen
cmp esi, 1
mov ebx, eax
~~jz short loc_401358~~
mov [esp+18h+var_18],...
mov ecx, 1

mov [esp+18h+var_14], ecx
call _printf
~~jmp short loc_40132F~~
mov [esp+18h+var_18], ebx
mov edx, 1
mov eax, offset aCmdDDDone
mov [esp+18h+var_10], edx
mov [esp+18h+var_14], eax

call _fprintf
mov ebx, [ebp+var_8]
mov eax, 1
mov esi, [ebp+var_4]

mov esp, ebp
pop ebp
retn
```

{i}

```
push ebp
mov ebp,esp
sub esp,28h
mov [ebp+var_C],ebx
mov eax, [ebp+arg_8]
mov ebx, [ebp+arg_0]
mov [ebp+var_4],edi
mov edi,offset unk_404000
mov [ebp+var_8],esi
xor esi,esi
mov [esp+28h+var_24],edi
mov [esp+28h+var_28],eax
call _fopen
cmp ebx,1
mov edi,eax
~~jz short loc_401370~~
mov [esp+28h+var_28],...
mov ebx,1
mov esi,4
mov [esp+28h+var_24],ebx
call _printf
~~jmp short loc_401339~~
mov [esp+28h+var_1C],esi
mov edx,1
mov eax,offset aCmdDDDone
mov [esp+28h+var_28],edi
mov [esp+28h+var_20],edx
mov [esp+28h+var_24],eax
call _fprintf
mov ebx,[ebp+var_C]
mov eax,1
mov esi,[ebp+var_8]
mov edi,[ebp+var_4]
mov esp,ebp
pop ebp
retn
```

{ii}

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:

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

and doing the same for G_2 results in the following:

$$(1', 2', 4'), (1', 2', 6^*), (1', 3', 5'), (2', 4', 5'), (2', 6^*, 7^*), (6^*, 7^*, 5').$$

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-

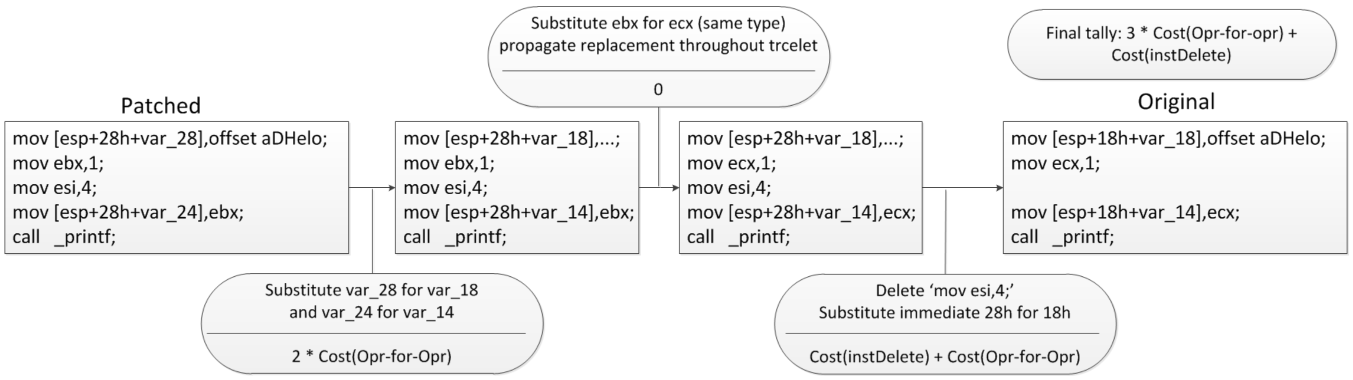

Substitute ebx for ecx (same type)
propagate replacement throughout trcelet
0
Final tally: 3 * Cost(Opr-for-opr) + Cost(instDelete)
Patched
Original
mov [esp+28h+var_28],offset aDHelo;
mov ebx,1;
mov esi,4;
mov [esp+28h+var_24],ebx;
call _printf;
mov [esp+28h+var_18],...;
mov ebx,1;
mov esi,4;
mov [esp+28h+var_14],ebx;
call _printf;
mov [esp+28h+var_18],...;
mov ecx,1;
mov esi,4;
mov [esp+28h+var_14],ecx;
call _printf;
mov [esp+18h+var_18],offset aDHelo;
mov ecx,1;
mov [esp+18h+var_14],ecx;
call _printf;
Substitute var_28 for var_18
and var_24 for var_14
2 * Cost(Opr-for-Opr)
Delete 'mov esi,4;'
Substitute immediate 28h for 18h
Cost(instDelete) + Cost(Opr-for-Opr)

Figure 4. A successful rewriting attempt with cost calculation

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 [instDelete].
- Add an instruction [instAdd].
- Substitute one operand for another operand of the same type [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 [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 $Cost$ 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 *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 (`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.

instr ::= *nullary* | *unary op* | *binary op op* | *trenary op op op*
op ::= `[` *OffsetCalc* `]` | *arg*
arg ::= *reg* | *imm*
OffsetCalc ::= *arg* | *arg aop OffsetCalc*
aop ::= `+` | `-` | `*`
reg ::= `eax` | `ebx` | `ecx` | `edx` | ...
nullary ::= `aad` | `aam` | `aas` | `cbw` | `cdq` | ...
unary ::= `dec` | `inc` | `neg` | `not` | ...
binary ::= `adc` | `add` | `and` | `cmp` | `mov` | ...
trenary ::= `imul` | ...

Figure 6. Simplified grammar for x86 assembly

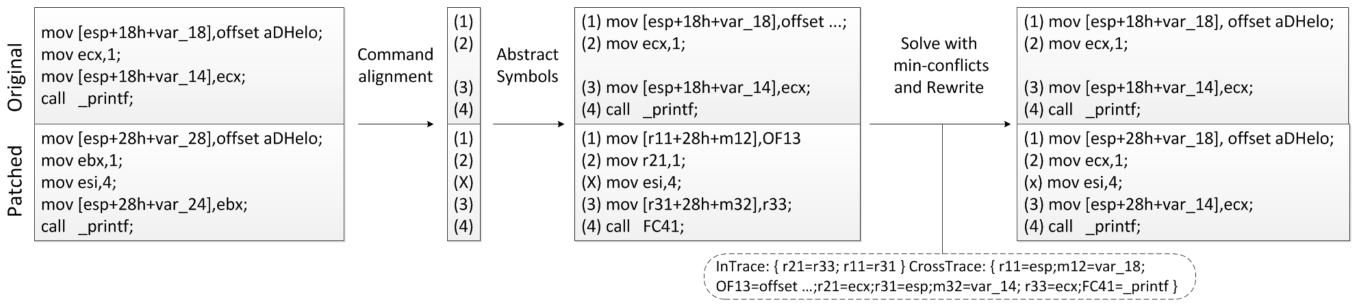
3. Preliminaries

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

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



```
Original:
mov [esp+18h+var_18],offset aDHelo;
mov ecx,1;
mov [esp+18h+var_14],ecx;
call  _printf;

Patched:
mov [esp+28h+var_28],offset aDHelo;
mov ebx,1;
mov esi,4;
mov [esp+28h+var_24],ebx;
call  _printf;
```

```
(1)
(2)

(3)
(4)

(1)
(2)
(X)
(3)
(4)
```

```
(1) mov [esp+18h+var_18],offset ...;
(2) mov ecx,1;

(3) mov [esp+18h+var_14],ecx;
(4) call  _printf;

(1) mov [r11+28h+m12],OF13
(2) mov r21,1;
(X) mov esi,4;
(3) mov [r31+28h+m32],r33;
(4) call  FC41;
```

```
(1) mov [esp+18h+var_18], offset aDHelo;
(2) mov ecx,1;

(3) mov [esp+18h+var_14],ecx;
(4) call  _printf;

(1) mov [esp+28h+var_18], offset aDHelo;
(2) mov ecx,1;
(x) mov esi,4;
(3) mov [esp+28h+var_14],ecx;
(4) call  _printf;
```

```
InTrace: { r21=r33; r11=r31 } CrossTrace: { r11=esp;m12=var_18;
OF13=offset ...;r21=ecx;r31=esp;m32=var_14; r33=ecx;FC41=_printf }
```

Figure 5. A full alignment and rewrite for the basic blocks 3 and 3' of Fig. 1(b) and Fig. 2(b)

an memory address offset (in this case, its value is read as part of the computation).

- write(inst) : $Instr \rightarrow 2^{Reg}$, the set of registers being written by the instruction $inst$.
- args(inst) : $Instr \rightarrow 2^{Arg}$, given an instruction returns the set of arguments that appear in the instruction.
- SameKind(inst,inst): $Instr \times Instr \rightarrow Boolean$, given two instructions returns true if both instructions have the same structure, meaning that they have the same mnemonic, the same number of arguments, and all arguments are the same up to renaming of arguments *of the same type*.

Following are some examples of running these functions:

#	Instruction	Args	Read	Write
inst1	add eax, ebx	{eax,ebx}	{eax,ebx}	{eax}
inst2	mov eax,[ebp+4]	{eax,ebp,4}	{ebp}	{eax}
inst3	mov ebx,[esp+8]	{ebx,esp,8}	{esp}	{ebx}
inst4	mov eax,[ebp+ecx]	{eax,ebp,ecx}	{ebp,ecx}	{eax}

Also, SameKind(inst2,inst3)=true, but SameKind(inst3,inst4)=false, as the last argument used in the offset calculation has different types in the two instructions (immediate and register respectively.)

Control Flow Graphs and Basic Blocks We use the standard notions of basic blocks and control-flow graphs. A basic block is a sequence of instructions with a single entry point, and at most one exit point (jump) at the end of the sequence. A control flow graph is a graph where the nodes are basic blocks and directed edges represent program control flow.

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.

One of the fundamental choices made in the process of constructing our similarity measure was the use of disassembly "symbols" (registers, memory offsets, etc.) to represent the code. This approach coincides with our need for flexibility, as we want to be able to detect the expected deviations that might be caused by optimizations and patching.

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{ii}):

- 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 `0x00404002`, containing the string "DONE", was replaced with `aCmdDDDone`
- 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 [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

```
   Input: T,R - target and reference functions,
          NM - normalizing method: ratio or containment
          k - tracelet size, $\alpha, \beta$ - threshold values
   Output: IsMatch - true if the functions are a match,
           SimilarityScore - the normalized similarity score
1  Algorithm FunctionsMatchScore(T,R,NM,k,$\alpha$,$\beta$)
2      RefTracelets = ExtractTracelets(R,k)
3      TargetTracelets = ExtractTracelets(T,k)
4      MatchCount = 0
5      foreach $r \in RefTracelets$ do
6          foreach $t \in TargetTracelets$ do
7              AlignedInsts = AlignTracelets(r,t)
8              t' = RewriteTracelet(AlignedInsts,r,t)
9              S = CalcScore(r,t')
10             RIdent = CalcScore(r,r)
11             TIdent = CalcScore(t',t')
12             if Norm(S,RIdent,TIdent,NM) $> \beta$ then
13                 MatchCount++
14         end
15     end
16     SimilarityScore = MatchCount $/|RefTracelets|$
17     isMatch = SimilarityScore $> \alpha$
```

The algorithm starts by decomposing each of the two functions into a set of k-tracelets (lines 2-3). We define the exact 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 cover 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-traclets 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 uses a helper function, `StripJumps`. This function takes a sequence of assembly instructions in which a jump may appear as the last instruction, and returns the sequence without the jump instruction.

Algorithm 2: Extract all k-tracelets from a CFG.

```
Input: G=$\langle B, E \rangle$ - control flow graph, k - tracelet size
Output: T - a list of all the tracelets in the CFG
Algorithm ExtractTracelets(G,k)
    $result = \emptyset$
    foreach $b \in B$ do
        result $\cup$= Extract(b,k)
    end
    return result
Function Extract(b,k)
    bCode = {stripJumps(b)}
    if $k = 1$ then
        return bCode
    else
        return $\bigcup_{\{b'|(b,b') \in E\}}$ 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 "**`r`**`orx` **`e`**`dx,es`**`i`**" with "`inc` **`rdi`**" (shown in bold, `rorx` is a rotate instruction, and `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 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;` is a letter) and use a specialized match value table, which is really a similarity measure between assembly instructions, to perform the tracelet alignment. It is important to note that in our method we give a high grade to perfect matches (instead of a 0 edit distance in the original method). For example, the score of comparing `push ebp;` with itself is 3, whereas the score of `add ebp,eax;` with `add esp,ebx;` is only 2. We compute the similarity between assembly instructions as follows:

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

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 pre-processed to allow for better matches (Section 4.1), while others, such as registers, are architecture dependent.

Algorithm 3: Calculate similarity score between two tracelets

```
 Input: T,R - target and reference tracelets
Output: Score - tracelet similarity score
1  Algorithm CalcScore(T,R)
2      A = InitMatrix(|T|,|R|)
3      // Access outside the array returns a large negative value
       for $i = |T|; i > 0; i - -$ do
4          for $j = |R|; j > 0; j - -$ do
5              $A[i,j]$ =Max(
6                  Sim($T[i]$,$R[j]$) + $A[i+1, j+1]$, // match
7                  $A[i+1, j]$, // insert
8                  $A[i, j+1]$ // delete
9              )
10         end
11     end
12     $Score = A[0,0]$
```

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: $S/min(RIdent, TIdent)$.
2. Ratio: taking into consideration the proportional size of the unmatched instructions in both tracelets.
   Calculation: $(S * 2)/(RIdent + TIdent)$.

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, $|args(r)| = |args(t)|$ holds (this is required for aligned instructions; see Section 4.3). For every pair of aligned arguments (aligned with regard to their linear position, eg., $eax \longleftrightarrow ebx, ecx \longleftrightarrow edx$, 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 φ (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.

Algorithm 4: Rewrite target code to match reference.

```
  Input: $AlignedInsts$ - a set of tuples with aligned
         reference and target assembly instructions (Note that
         aligned instructions have the same number of
         arguments), T,R - target and reference tracelets
  Output: T' - rewritten target code
1 Algorithm RewriteTracelet (T,R)
2     foreach $(t, r) \in AlignedInsts$ do
3         for $i = 1; i < |args(t)|; i++$ do
4             $s_t = args(t)[i]$
5             $s_r = args(r)[i]$
6             $nv = newVarName(s_t)$
7             $\varphi = \varphi \wedge (nv = s_r)$
8             if $s_t \in read(t)$ and $lastWrite(s_t) \neq \perp$ then
9                 $\varphi = \varphi \wedge (nv = lastWrite(s_t))$
10            else if $s_t \in write(t)$ then
11                $lastWrite(s_t) = nv$
12        end
13    end
14    $vmap = solve(\varphi, symbols(R))$
15    foreach $t \in T$ do
16        t' = t
17        foreach $s_t \in t$ do
18            if $(s_t) \in vmap$ then
19                $t' = t'[s_t \mapsto vmap(s_t)]$
20            end
21        end
22        $T'.append(t')$
23    end
```

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 stripped 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

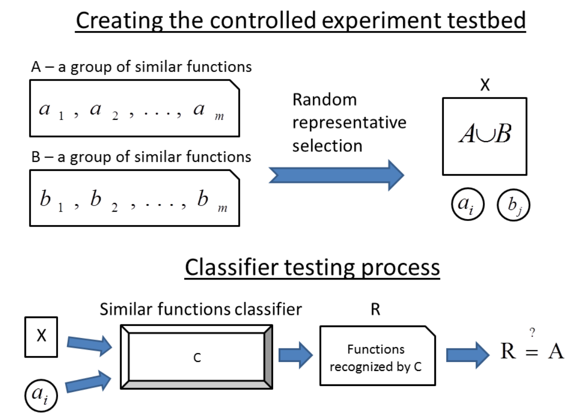

Creating the controlled experiment testbed
A – a group of similar functions
$a_1, a_2, \ldots, a_m$
B – a group of similar functions
$b_1, b_2, \ldots, b_m$
Random representative selection
X
$A \cup B$
a_i
b_j
Classifier testing process
Similar functions classifier
X
a_i
C
R
Functions recognized by C
$R \overset{?}{=} A$

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

when solving the constraint we are willing to drop conjuncts if the full constraint is not satisfiable. The constraint solving process uses a backtracking solver. This process is bounded to 1000 backtracking attempts and returns the best assignment found, with minimal conflicts. The assignment is then used to rewrite the abstracted arguments in all of the matched instructions and, for completeness, a cache of the argument swaps is used to replace the values in the deleted instructions. When this process is complete, the tracelets's match score will be recalculated (using the same algorithm) to get the final match score. Arguably, after this process, the alignment of instructions might change, and repeating this process iteratively might help improve the score, but experiments show that subsequent attempts after the first run are not cost effective.

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 `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 β). 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

K	# Tracelets	# Compares	$\frac{\#Tracelets}{Function}$	$\frac{\#Instructions}{Tracelet}$
k=1	$229,250$	$1.586 * 10^8$	12.839[39.622]	5.738[21.926]
k=2	$211,395$	$1.456 * 10^8$	11.839[39.622]	11.091[23.768]
k=3	$188,133$	$1.284 * 10^8$	10.536[39.445]	16.518[39.445]
k=4	$166,867$	$1.139 * 10^8$	9.345[39.096]	22.072[29.965]
k=5	$147,634$	$1.008 * 10^8$	8.268[38.484]	27.618[31.021]

Table 1. Test-bed statistics. For average values standard deviation is presented in square brackets.

was done to avoid over matching of too-small functions in the containment calculation.

Our general testing process is shown in Fig. 7. From each group of similar functions we chose a random representative and then tested it against the entire testbed. We required our similarity classifier to *find all similar functions* (the other functions from its original similarity group) with high precision and recall.

In the second stage we expanded each test group with executables downloaded from different internet repositories (known to contain true matches, i.e., functions similar to the ones in the group), and added a new "noise" group containing randomly downloaded functions from the internet. At this stage we also implicitly added executables containing functions that are both in a different context *and* had undergone code change (different version). An example for such samples is shown in Sec. 6.2.

Test-bed statistics In the final compositions of our testbed (at the beginning of the second stage described above), we had a total of *over a million* functions in our database. At that point we gathered some statistics about the collected functions. The gathered information gives an interesting insight into the way modern compilers perform code layout and the number of computations that were performed. These statistics are shown in Table 1. A slightly confusing statistic in this table is the number of tracelets as a function of k. The number of tracelets is expected to grow as a function of their length (the value of k) in "ordinary" graph structures such as trees, where nodes might have a high out-degree. This is not the case for a structure such as a CFG. The average in-degree of a node in a CFG is $0.9221(STD = 0.2679)$, and the average out-degree is $0.9221(STD = 1.156)$. This, in addition to our omitting paths shorter than k when extracting k tracelets, caused the number of tracelets to decline for higher values of k. The number of instructions in a tracelet naturally rises, however. Note the *very high* number of compare operations required to test similarity against a reasonable datebase, and that the average size of a basic block (or a 1-tracelet) is 6 instructions. Fortunately, this process can be easily parallelized (Sec. 5.2).

Evaluating classifiers The challenge in building a classifier is to discover and maintain a "noise threshold" across experiments, where samples scoring below it are not classified as similar.

The receiver operating characteristic (ROC) is a standard tool in evaluation of threshold based classifiers. The classifier is scored by testing all of the possible thresholds consecutively, enabling us to treat each method as a binary classifier (outputting 1 if the similarity score is above the threshold). For binary classifiers, accuracy is determined using the True Positive (TP, the ones we know are positive), True Negative (TN), Positive (P, the ones classified as positive) and Negative (N, the ones classified as negative):

$$Accuracy = (TP + TN)/(P + N).$$

Plotting the results for the different thresholds on the same graph yields a curve; the area under this curve (AUC) is regarded as the accuracy of the proposed classifier.

CROC is a recent improvement of ROC that address 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 aforemention 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 graphs, 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 tracelets, $(1, 2, 3)$ 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 recalculation 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" (β in Algo. 1), a threshold parameter above which a tracelet will count as a match for another.

β Value	$10-20$	30	40	50	60	$70-90$	100
AUC[CROC]	0.15	0.23	0.45	0.78	0.95	0.99	0.91

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

- The "function coverage rate match barrier" (α 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 α, for every reasonable value of k ($1-4$), we ran 10 experiments testing all values of β (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. 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).

	n-grams Size 5,Delta 1	Graphlets K=5	Traclets K=3	
			Ratio	Contain
AUC[ROC]	0.7217	0.5913	1	1
AUC[CROC]	0.2451	0.1218	0.99	0.99

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.

Table 3 summarizes the results of 6 different, carefully designed experiments, using the following representatives:

- `quotearg_buffer_restyle` function from `wc` v6.12 (a library function used by multiple Coreutils applications).
- The same function from `wc` v7.6 "implanted" in `wc` v8.19.
- `getftp` from `wget` v1.10.
- The vulnerable function from `tar` (described above).
- Two random functions selected from the "noise group".

Each experiment considers a single query (function) against the DB of *over a million* examples drawn from standard Linux utilities. For each executables group, the true positives (the similar functions) were manually located and classified as such, while the rest were assumed to be true negatives. Although the same function should not appear twice in the same executable, every function which yielded a high similarity grade was manually checked to confirm it is a false positive.

The challenge is in picking a *single* threshold to be used in all experiments. Changing the threshold between queries is not a realistic search scenario.

Each entry in the table is the area under the curve, which represents the best-case match score of the approach. ROC curves (and their improvement CROC) allow us to compare different classifiers in a "fair" way, by allowing the classifier to be tested with all possible thresholds and plotting them on a curve.

For example, the value 0.7217 for ROC with n-grams (size=5, delta=1) yields the the best accuracy (see definition in Section 5.1) obtained for any choice of threshold for n-grams. In other words, the best accuracy achievable with n-grams with these parameters is 0.7217, in contrast to 0.99 accuracy in our approach. This is because n-grams and graphlets use a coarse matching criterion not suited for code changes. This could also be attributed to the fact that they were targeted for a different classification scenario, where the goal is to find only one matching function, whereas we strive to find *multiple* similar functions. (The former scenario requires that the matched function be the top or in the top 10 matches.) When running an experiment in that scenario, the other methods do get better, though still inferior, results ($\sim$ 90% ROC, $\sim$ 50% CROC).

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).

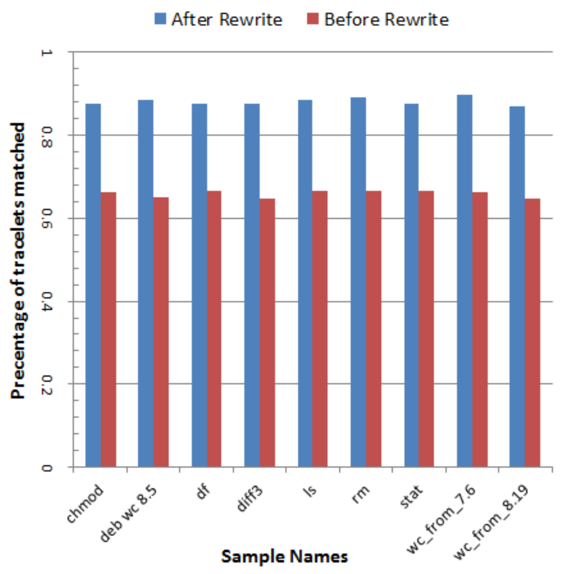


Figure 8. Matching a function across different contexts and after code changes

- 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

Item	Op	Time (secs)			
		AVG (STD)	Med	Min	Max
Tracelet	Align	0.0015 (0.0022)	0.00071	0.00024	0.092
	Align & RW	0.0035 (0.0057)	0.0025	0.00052	0.23
Function	Align	3.8(12.2)	1.6	0.076	627
	Align & RW	8.6(25.78)	2.8	0.087	1222

Table 4. Runtimes of tracelet-to-tracelet and function-to-function comparisons, with and without the rewrite engine.

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 executable behavior and detect anomalies which could be attributed to malware and used to identify it. Employing dynamic measures in this case enables bypassing malware defences such as packing. We employ a static approach, and dealing with executable obfuscation is out of the scope of our work.

8. Limitations

During our experiments, we observed some limitations of our method, which we now describe.

Different optimization levels: We found that when compiling source code using `O1` optimization level, the resulting binary can be used to find `O1`,`O2` and `O3` versions. However, `O0` and `Os` are very different and are not found. Nonetheless, when we have the source code for the instance we want to find, we can compile it with all the optimization levels and search them one by one.

Cross-domain assignments: A common optimization is replacing an immediate value with a register already containing that value. Our method was designed so that each symbol can only be replaced with another in the same domain. Our system could also search cross-domain assignments, but this would notably increase the cost of performing a search. Further, our experiments show very good precision even when this option is disabled.

Mnemonic substitution: Because our approach requires compared instructions' mnemonics to be the same in the alignment stage, if a compiler were to select a different mnemonic the matching process would suffer. Our rewrite engine could be extended to allow common substitutions; however, handling the full range of instruction selection transformations might require a different approach.

Matching small functions: Our experiments use functions with a minimum of 100 basic-blocks. Attempting to match smaller functions will often produce bad results. This is because we require a percent of the tracelets to be covered. Some tracelets are very common (leading to false positives) while slight changes to others might result in major differences that cannot be evened out by the other tracelets. Furthermore, small functions are sometimes inlined.

Dealing with inlined functions: This is a problem in two cases, when the target function was inlined, and when functions called inside the reference functions are inlined into it. Some of these situations could be handled — but only to certain extent — the containment normalization method.

Optimizations that duplicate code: Code duplication for avoiding jumps, for example loop unrolling. Similarly to inlined functions, our method can manage these optimizations when the containment normalization method is used.

9. Conclusions

We presented a new approach to searching code in executables. To compute similarity between functions in stripped binary form, we decompose them 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

[1] A heap based vulnerability in gnu's rtapelib.c. `http://www.cvedetails.com/cve/CVE-2010-0624/`.

[2] Hex-rays IDAPRO. `http://www.hex-rays.com`.

[3] Yard-plot. `http://pypi.python.org/pypi/yard`.

[4] Balakrishnan, G., and Reps, T. Divine: discovering variables in executables. In *VMCAI'07* (2007), pp. 1–28.

[5] Ball, T., and Larus, J. R. Efficient path profiling. In *Proceedings of the 29th Int. Symp. on Microarchitecture* (1996), MICRO 29.

[6] Bansal, S., and Aiken, A. Automatic generation of peephole superoptimizers. In *ASPLOS XII* (2006).

[7] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. Comparison and evaluation of clone detection tools. *IEEE TSE 33*, 9 (2007), 577–591.

[8] Bruschi, D., Martignoni, L., and Monga, M. Detecting self-mutating malware using control-flow graph matching. In *DIMVA'06*.

[9] Comparetti, P., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., and Zanero, S. Identifying dormant functionality in malware programs. In *IEEE Symp. on Security and Privacy* (2010).

[10] Horwitz, S. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90*.

[11] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. In *PLDI '88* (1988).

[12] Jang, J., Woo, M., and Brumley, D. Towards automatic software lineage inference. In *USENIX Security* (2013).

[13] Khoo, W. M., Mycroft, A., and Anderson, R. Rendezvous: a search engine for binary code. In *MSR '13*.

[14] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. Polymorphic worm detection using structural information of executables. In *Proc. of int. conf. on Recent Advances in Intrusion Detection*, RAID'05.

[15] Myles, G., and Collberg, C. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pp. 314–318.

[16] Partush, N., and Yahav, E. Abstract semantic differencing for numerical programs. In *SAS* (2013).

[17] Reps, T., Ball, T., Das, M., and Larus, J. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC '97/FSE-5*.

[18] Rosenblum, N., Zhu, X., and Miller, B. P. Who wrote this code? identifying the authors of program binaries. In *ESORICS'11*.

[19] Rosenblum, N. E., Miller, B. P., and Zhu, X. Extracting compiler provenance from program binaries. In *PASTE'10*.

[20] Saebjornsen, A., Willcock, J., Panas, T., Quinlan, D., and Su, Z. Detecting code clones in binary executables. In *ISSTA '09*.

[21] Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. In *ASPLOS '13*.

[22] Sharma, R., Schkufza, E., Churchill, B., and Aiken, A. Data-driven equivalence checking. In *OOPSLA'13*.

[23] Singh, R., Gulwani, S., and Solar-Lezama, A. Automated feedback generation for introductory programming assignments. In *PLDI '13*, pp. 15–26.

[24] Swamidass, S. J., Azencott, C.-A., Daily, K., and Baldi, P. A CROC stronger than ROC. *Bioinformatics 26*, 10 (May 2010).

[25] Wagner, R. A., and Fischer, M. J. The string-to-string correction problem. *J. ACM 21*, 1 (Jan. 1974), 168–173.