Leveraging Semantic Signatures for Bug Search in Binary Programs

Jannik Pewny[†], Felix Schuster[†], Christian Rossow[‡], Lukas Bernhard[†], Thorsten Holz[†]
[†]Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
[‡]Cluster of Excellence, MMCI
Saarland University, Germany

ABSTRACT

Software vulnerabilities still constitute a high security risk and there is an ongoing race to patch known bugs. However, especially in closed-source software, there is no straightforward way (in contrast to source code analysis) to find buggy code parts, even if the bug was publicly disclosed.

To tackle this problem, we propose a method called TREE EDIT DISTANCE BASED EQUATIONAL MATCHING (TEDEM) to automatically identify binary code regions that are "similar" to code regions containing a reference bug. We aim to find bugs both in the same binary as the reference bug and in completely unrelated binaries (even compiled for different operating systems). Our method even works on proprietary software systems, which lack source code and symbols.

The analysis task is split into two phases. In a preprocessing phase, we condense the semantics of a given binary executable by symbolic simplification to make our approach robust against syntactic changes across different binaries. Second, we use *tree edit distances* as a basic block-centric metric for code similarity. This allows us to find instances of the same bug in different binaries and even spotting its variants (a concept called *vulnerability extrapolation*). To demonstrate the practical feasibility of the proposed method, we implemented a prototype of TEDEM that can find real-world security bugs across binaries and even across OS boundaries, such as in `MS Word` and the popular messengers `Pidgin` (Linux) and `Adium` (Mac OS).

1. INTRODUCTION

Security vulnerabilities in software systems have plagued us since many years. There are numerous classes of security-relevant bugs such as buffer overflows, use-after-free conditions, and logical flaws. While it is possible to define a bug class on an abstract level (e. g., "a copy operation writes data beyond array bounds"), the concrete manifestation of a vulnerability in a program can be arbitrarily complex. As such, it is hard to apply the abstract definitions in practice.

Many approaches were proposed that perform an automated security analysis of a given piece of (binary or source) code to detect specific security vulnerabilities. Such methods either leverage characteristic patterns for identifying security vulnerabilities [10] or utilize program analysis techniques such as fuzzing [24], taint analysis [22, 25], and symbolic execution [3, 4] to find bugs. In general, the problem of identifying all bugs in a given program is undecidable and thus all approaches have certain shortcomings. For example, they can only detect predefined kinds of vulnerabilities or miss some manifestations of a bug (i.e., they have false negatives) during the analysis phase.

ACSAC December 8-12, 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s).
Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664269.

Finding similar bugs. A variant of the general problem of finding bugs is to identify bug doublets, given a concrete example of a buggy piece of code. The problem can be stated as follows: *Given an exemplary bug, find locations in a given program which have this or a similar bug.*

While finding completely new bugs is certainly the more ambitious goal, there is a great benefit in finding *similar* bugs. For example, it helps to find dormant bugs, identify false negatives, or to check if a binary supplied by a third-party still has a known bug or is patched already. Yamaguchi et al. [28] studied this problem on the source code level and they coined the term *vulnerability extrapolation*. The underlying assumption is the following: When code is similar to code which contains a bug, it is likely to contain the same bug as well [21]. Based on this assumption, the problem of vulnerability extrapolation can be addressed with a reasonable code similarity metric.

In the real world, there are many reasons for bug doublets. The underlying root cause is typically *code reuse*, which may happen due to copy'n'paste or project forks [17]. When maintaining a code base with different versions of the same code, it is almost certain that a bug found in one version of a program is not fixed in all versions. This problem grows when the code is not maintained by the same group of people or when code is shared among different projects (e. g., an open-source library). Copied code may be modified to suit the slightly different needs and bugs tend to happen in similar, but not equal, circumstances. Thus, looking for *exact* matches may severely limit usefulness.

When looking for bugs, there are many reasons to analyze binary code instead of source code. Most importantly, source code is not always available (e. g., proprietary code used in commercial products, third-party libraries, or legacy code). In addition, the compilation phase itself may introduce bugs that simply were not present in the source code [27]. Unfortunately, finding bugs gets a lot harder when the analysis is performed on binary code. Several techniques can detect code reuse in source code [12, 14, 18], but they cannot directly be applied to binary code: Transformations and optimizations performed by a compiler such as register assignments, function inlining, constant folding/propagation, or instruction reordering alter code structure and code syntax. Hence, identifying similar code regions is challenging.

Our approach. In this paper, we introduce a method called Tree Edit Distance based Equational Matching (TEDEM). Given a so called *signature*, TEDEM finds code locations in a binary that has similar bugs. TEDEM finds vulnerabilities in binary programs by identifying buggy code based on semantic signatures for a given security bug. In contrast to source code-based techniques that heavily rely on symbols (neglecting semantic information), we take another approach to address the problem. In a preprocessing phase, we first disassemble a binary and extract semantic information of the basic blocks in forms of *expression trees* (i. e., equations that summarize the results of the computations performed in the basic block). This symbolic representation makes our analysis robust against small syntactic changes across binaries.

Based on this information, we can search for similar code regions. To this end, we introduce a metric for fine-grained, basic block-centric comparison of binary code, which allows us to find bugs through code similarity. The metric takes most of the semantic information into account. In contrast, methods focusing solely on mnemonics [20], CFG structure [15], or API calls [1] are by design ignoring certain important semantic information. Thus, they often cannot reveal sufficient details to compare small code segments, as it is especially required for extrapolating vulnerabilities. If the granularity is too coarse, one can only look for aggregated, statistical effects over bigger structures, like functions or libraries — losing the flexibility to search for similar bugs contained in other substructures.

We evaluated TEDEM on multiple large and real-world binaries. For example, we found unpatched vulnerabilities in `PuTTY` forks across binary boundaries and in `Adium`'s libpurple even across OS boundaries. Furthermore, we found patch level discrepancies in `MS Office` products. Despite its fine-granular bug search, TEDEM scales reasonably well and could find bugs even in large programs such as `MS Word 2013` (6.3 MB and 416,736 basic blocks) in less than 18 min.

Contributions. In this paper, we make the following three main contributions:

- We designed and implemented a basic block-centric metric based on tree edit distances to measure the similarity of two pieces of binary code that is capable of tolerating small syntactic changes in the program.
- Using this as a building block, we implemented a scheme to measure similarity of larger code constructs, like sub-CFGs and functions.
- Empirical measurements demonstrate the viability of the approach and we found real-world bugs across binary and even across OS boundaries.

2. RELATED WORK

Finding bugs in software systems has attracted a lot of research. In the following, we briefly review prior work in this area and discuss how it relates to our approach.

2.1 Code Clones in Source Code

ReDeBug [11] is a source code-based system to identify unpatched code clones in a very fast way, which is due to the fact that they only search for close-to-exact matches. While it shares the goal with our work, it requires source code and is thus of no help for binary software.

Yamaguchi et al. proposed *extrapolation of vulnerabilities* on source code level [28]. Given source code written in C, they extract *abstract syntax trees* (AST) of all contained functions. Based on the properties of their ASTs, functions are projected into a high-dimensional vector space. Subsequently, methods from the realm of machine learning are applied in order to cluster functions, which reveals functions similar to ones known to be vulnerable. We achieve comparable results as reported by Yamaguchi et al. (cf. Section 5), just that our approach does not require source code.

2.2 Binary Code Comparison

BinDiff, as proposed by Dullien and Rolles [5], is the de facto standard commercial tool for comparing two pieces of binary code. *BinDiff* mainly relies on the structural similarity and makes only little use of code semantics. Thus, one of its weaknesses is that it conceptually struggles with matching functions that are only invoked indirectly and are thus not connected in a call graph (regularly the case for, e. g., C callbacks, C++ virtual methods, Objective-C methods). Furthermore, BinDiff regularly mismatches basic blocks on the foundations of similar structure, but different semantics.

Another group of related work explicitly assesses *semantic* similarities/differences of binary software. Many works are based on summarizing a basic block's effects on the program state as a set of equations (sometimes referred to as *symbolic execution* [25]) and we also utilize such an approach. *BinHunt* [7] tries to find semantic differences between a binary program and its patched version to pinpoint vulnerabilities. To determine how similar two basic blocks are, they test all possible pairs of equations from both sets for equality using a theorem prover. Based on identified semantically equal basic blocks, a custom backtracking algorithm finds the largest maximum common induced subgraph between two functions and derives the similarity between two functions from this. Ming et al. proposed *iBinHunt* [19], which adds taint analysis to *BinHunt* to allow for less basic block comparisons. As these tools are targeted for whole-binary, function-level "diffing", they are somewhat ill-suited for our use-case of finding small and fine-grained bug signatures: their idea would fail with function inlining and could not find out-of-context re-use of vulnerable code snippets. In addition, these two approaches can only verify if code is "equal" and have to rely on handcrafted heuristics to use their non-continuous similarity score as a metric. As we will show, a proper metric is vital in the context of bug searches.

BinHash is a sampling-based approach to identify semantically similar binary functions [13]. *BinHash* evaluates the sets of basic block equations concretely for randomly-chosen input vectors and all their permutations. Functions are then compared via locality-sensitive hashing over the output vectors of their basic blocks. Since there are generally $n!$ permutations for a given vector, scalability problems arise for basic blocks with large input dimensions. While evaluating TEDEM, we encountered basic blocks with an input dimension even larger than 50, which demonstrates scalability issues with approaches such as *BinHash*.

Lakhotia et al. presented *BinJuice* [16]. They normalize a basic block's equation set in several steps to extract its semantic "juice". Semantically similar basic blocks are then identified through lexical comparisons of juices. Since the authors did not give a quantifiable evaluation, it remains open if their metric is suitable. In addition, the authors did not expand their similarity computation beyond the scope of individual basic blocks, which is required for bug search.

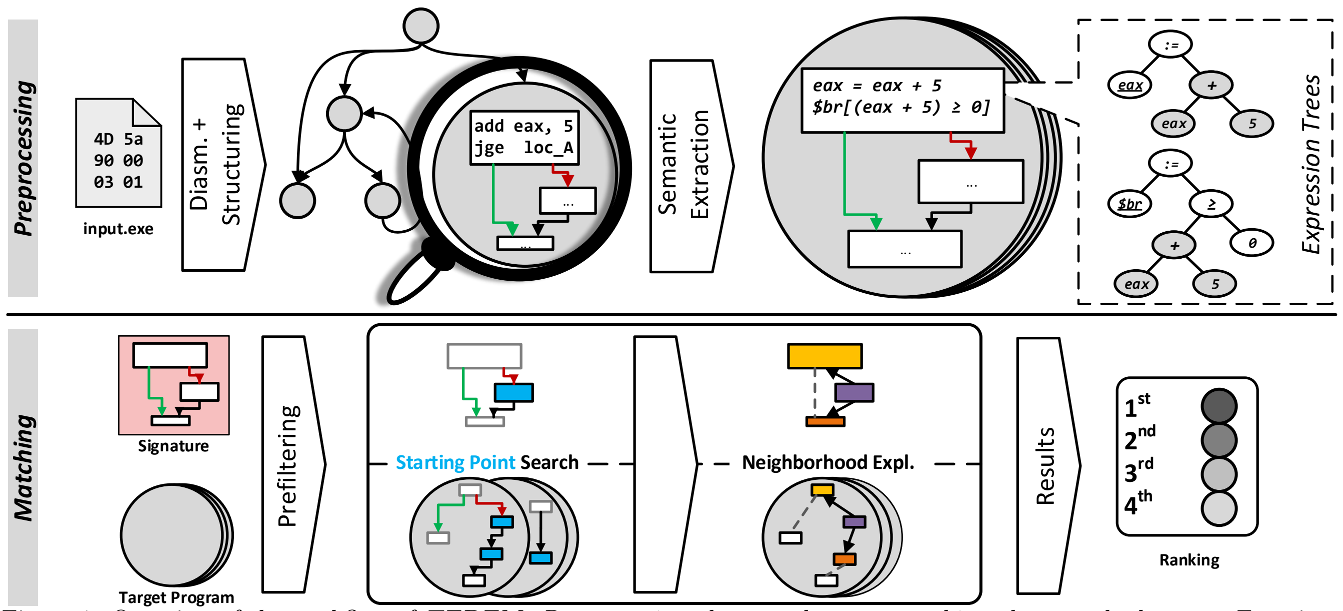

Figure 1: Overview of the workflow of TEDEM; *Preprocessing phase* at the top, *matching phase* at the bottom. Functions are gray circles, basic blocks are solid white rectangles.

Exposé [23] identifies library re-use in binary software. First, promising function matches between the library and a binary program are gathered by examining simple features like *number of arguments* or *cyclomatic complexity*. Second, candidate function matches are verified by symbolically executing both functions and leveraging a theorem prover. As *Exposé* aims to identify library matches, it is too coarse-grained for certain scenarios when extrapolating bugs.

2.3 Finding Unknown Bugs

Tools such as AEG [2] and Mayhem [3] aim at automatically detecting security vulnerabilities. They use binary analysis techniques such as symbolic/concolic execution to reason about vulnerable code paths. Our goal is different in the sense that we search for similar security vulnerabilities given an exemplary bug signature. In addition, many such approaches aim to find specific bug classes, such as control flow hijack vulnerabilities. In contrast, our system can be used to find any type of bug.

3. SYSTEM OVERVIEW

In the typical use case of our approach, an analyst knows about a concrete bug (maybe from a CVE advisory or from a patch) and aims to find similar bugs. In our terminology, the analyst extracts a *bug signature* and uses it to find further similar bugs in a *target program*.

Figure 1 visualizes the two phases of our TREE EDIT DISTANCE BASED EQUATIONAL MATCHING (TEDEM). First, in the upper half, we preprocess program binaries for our bug-search. This phase extracts disassembly and structure (i. e., a control flow graph) from the programs and then summarizes semantics of the basic blocks in form of expression trees. Preprocessing only has to happen once per signature and — independently — once per target program. The preprocessed output can be reused in future bug searches.

Second, as shown in the lower half of Figure 1, our system searches for a signature within a target program and identifies binary code parts which are similar to the signature. For this, we identify single high-quality basic blocks matches in the signature and the target program. We use these pairs of basic blocks as starting points to compare the full signature with parts of the target program, such as functions or parts thereof. At the end of the workflow, TEDEM reports the similarity of each function from the target programs to the signature in a sorted list, revealing the functions with substructures similar to the signature.

3.1 Program Preprocessing

In practice, compilers complicate the process of binary comparison, even if the binaries have the same source code base. Two different compilers or compiler configurations typically create different versions of binary code from equal source code. Most compilers apply memory or performance optimizations so that the representations between compiled binaries of two configurations differ significantly. For example, the usage of registers can change, different calling conventions may be used, the control flow graph may be altered, or arithmetic operations could be tweaked to optimize performance. This can occur if another compiler (or version) is used, the optimization levels are changed, or even if slightly modified source code allows the compiler to perform better optimizations. All these modifications complicate the process of comparing two binaries at the disassembly level.

In this section, we describe the preprocessing steps that we take to compensate for these issues.

3.1.1 Disassembly and Structuring

First, we obtain a disassembly of the programs, i. e., the original (buggy) program and the target program. We then split each program into *strict basic blocks*. That is, we extract the basic blocks and further split these at function calls. Each strict basic block is thus only left at its final instruction, the terminator, and only its first instruction is a jump target. While this definition is used for basic blocks as well, the instruction after a call is an implicit target of a return instruction and is naturally not the first instruction of a basic block. For the rest of this paper, whenever we mention a basic block, we refer to a *strict* basic block.

3.1.2 Semantic Extraction

Next, we transform the sequence of a basic block's instructions into equations. Each equation is an assignment where left-hand side and right-hand side both contain arbitrary computations. The left side of the equation is (or results in) a register or memory location. The equations are built such that each register, write-address, or jump condition only appears once on the left side of an assignment (like a single static assignment). We compute the equations by accumulating the computational steps performed on one such variable on the right side of the assignment. All inputs on the right side of the equations are not modified by preceding equations. For example, we consider the *input* variable `eax` as constant, even if a value is assigned to the *output* variable `eax` before the variable is used in a subsequent equation. Therefore, for a given input state, the equations can be evaluated independently from one another.

This preprocessing abstracts from instruction ordering and allows a natural sorting of the assignments. We chose an arbitrary but deterministic order to sort the equations, such that two equal basic blocks have the same order of equations.

Further, we add equations for the number of successors/predecessors. We also add a condensed terminator-equation, where we encode possible jump destinations and, if present, a branching condition. In addition, we distinguish various kinds of constants by their type of usage, e. g., code locations, data offsets, or arithmetic/logic constants.

We then simplify and normalize the assignments on a basic block level. For details, we refer to Section 4.1.2.

3.2 Bug Signature Matching

Once the binaries have been preprocessed, we continue with the original goal to find bugs in these programs. In this section, we define the notion of a *signature* (Section 3.2.1), which is searched for in the *target program*. We start by describing how we compare a basic block of the signature with a basic block of the target program (Section 3.2.2). We then outline our approach to prefilter basic blocks to gain search speedups, without decreasing effectiveness (Section 3.2.3). Lastly, we leverage the CFG to apply the signature (Section 3.2.4) on the target program.

3.2.1 Bug Signature

A *bug signature* describes a program part that is characteristic for a certain bug. We define a bug signature as *a subset of the CFG of a vulnerable function*. This means that an analyst only needs to identify the bug-relevant parts of a function. Essentially, the bug signature is binary code, i.e., contains basic blocks and their transitions and thus undergoes the same handling as the target programs. In the simple case of function-level code duplicates, the signature can be equal to the CFG of the vulnerable function. In more advanced bugs, more fine-grained signatures can be as small as parts of the equations of certain basic blocks. We will demonstrate in the evaluation section that – once a bug is known and has been understood – deriving a signature is straightforward with minimal manual effort.

However, too generic signatures risk to cause false positives. Therefore, instead of searching for entire bug classes, the signature focuses on finding specific bugs. For example, instead of only searching for writes to memory with lacking bound checks (to capture buffer overflows), we also search for contextual information that is specific to the buggy reference program (e.g., basic blocks surrounding the bug, concrete buffer lengths, and the specific way the memory is modified). Our search is thus *contextual*, as the signature is derived from a concrete bug context in a reference program. Despite the need for context, our approach can also find similar bugs in programs other than the reference program.

3.2.2 Basic Block Comparison

Given the signature, we want to find similar bugs in the target program. A cornerstone in this process is comparing basic blocks of the signature with basic blocks of the target program. We leverage the tree structure of the basic block equations for this comparison. After the preprocessing phase (Section 3.1), each basic block is represented as a list of equations. An equation is a tree, where the root node is an assignment or a basic block terminator, the leaf nodes are registers or constants (e. g., numbers, addresses), and intermediate nodes are operations. We thus leverage a tree comparison algorithm to measure similarity between two given basic blocks.

We chose to use a tree edit distance [26] (TED) to compare equations. The TED measures the minimum costs for transforming one tree (e. g., basic blocks from the signature) to another (e. g., basic blocks from the target program). That is, it measures the costs for modifying trees via node replacements and insertions/deletions of subtrees. We get low distances for similar, and higher distances for less similar inputs. The TED thus measures the syntactic difference in the condensed semantic representation of the basic block. The costs correlate with the number of nodes that need to be replaced, inserted or deleted, and are *at most* as large as the number of modified nodes. Tekli et al. provide a detailed description of the distance computation [26].

3.2.3 Prefiltering & Candidate Search

Naïvely, we could compare all basic blocks from the signature with all those from the target program and expand our search from there. However, especially if fine-grained similarity measures (as the TED) are used, this results in scalability problems. We thus leverage the fact that some basic blocks in the signature are more characteristic than others. For example, basic blocks that are too generic have likely similar counterparts spread among multiple parts of the program, and thus are not really useful to characterize a bug. Similarly, compared to a reference basic block in the signature, most of the basic blocks in the target program differ so much that fine-grained comparisons (e. g., using the TED) are not needed to tell that basic blocks are not similar. Guided by these observations, we (i) focus on the parts in the signature that are distinctive enough to serve as a marker for a bug, and (ii) reduce the number of basic block comparisons in the target program.

We found that a few *coarse-grained* basic block attributes are sufficient to indicate very low similarity of two basic blocks. That is, the coarse-grained similarity measure correlates with the fine-grained similarity computed via the TED. Coarse-grained basic block attributes are, for example, the number of equations, the depth of the equation trees, or the number of nodes in the tree. We thus evaluated if we can use these attribute-level comparisons in a potential prefiltering step. In particular, we measured how well these attributes reflect our notion of basic block similarity. We evaluated this in three binary programs (`Adium` on Mac OS, `ImageMagick` on Linux, `MS Word 2013` on Windows) by computing the correlation between the similarity of single coarse-grained attributes (e. g., number of equations in basic block A and B) and the fine-grained TED similarity score between the basic blocks. We found strong correlation coefficients (averaged over the binaries 0.63, 0.92, 0.52, respectively) for all three basic block attributes and thus use them as prefilters.

Our basic block prefiltering works as follows. First, we identify *characteristic* basic blocks from the signature to use them as *starting points* for our bug search. That is, we focus on those basic blocks in the signature whose attributes are equal to at most t percent of the basic blocks in the target program. We denote the resulting set of starting points in the signature as SP. The lower the threshold t, the more basic blocks are filtered and, thus, the more efficient is the bug search, while the risk to miss relevant basic blocks increases. We use $t = 5\,\%$ in our experiments to reasonably trade off accuracy against scalability.

Then, for all basic blocks in SP, we search for *matching candidates* in the target program. That is, given a signature starting point $sp_i \in SP$, we calculate the fine-grained similarity (using the TED) to all basic blocks in the target program and choose the n basic blocks with the lowest distances. We denote this set of matching candidates as C_i, where i corresponds to the index of starting point sp_i. The lower n, the less often we need to invoke later stages of the bug signature matching, i. e., the lower the runtime. However, with lower n the risk to miss a good match rises. We chose $n = 20$ in our evaluation since we found in an empirical test that this value provides a good trade-off in practice.

We use the resulting set of starting points (and their matching candidates) in the following section to measure the signature's similarity to the target program (instead of only comparing single basic blocks).

3.2.4 Neighborhood Exploration

So far we have discussed how we compare binary code at the level of basic blocks. However, in order to find bugs, comparing individual basic blocks is not sufficient, as multiple basic blocks are typically characteristic for a bug. Furthermore, the *structure* of code is often also characteristic for a bug. Consider for example a typical buffer overflow bug where a certain basic block performing boundary check may be missing inside a memory-writing loop. Therefore, we further explore the CFG of a function in the neighborhood of individual basic block matches with the bug signature.

We do so in a CFG-driven, greedy, but locally optimal manner: In step i), we choose a pair of starting point ($sp_i \in SP$) and one of its matching candidates ($c_j \in C_i$) in the target program to initiate our search. We use the pair (sp_i, c_j) as the initial match, i. e., our algorithm considers it as a fixed pair and will not change their mapping. Once the algorithm terminates for this concrete pair, it is repeated for all other matching candidates of this starting point. In addition, we will repeat the algorithm with multiple starting points, i. e., using different initial pairs.

Then, in step ii), we compare the adjacent neighbors of the pair that has been fixed in step i). Adjacency in this context means that the basic blocks are either CFG-wise successors or predecessors of the fixed. When matching, we take the direction of the CFG into account, i. e., we do not compare preceding basic blocks with succeeding basic blocks, and vice versa. To this end, we compute the distances of all adjacent basic blocks. We then use a matching algorithm (the Hungarian algorithm [6]) to find ideal mappings between basic blocks adjacent to the previously fixed pair.

At this point, each optimally mapped pair of basic blocks (one in the signature, one in the target program), becomes a new candidate for *broadening* the overall match. We keep track of these candidates with a priority queue, sorted along the distance between the mapped pair. In step iii), we pick the best candidate pair (i. e., lowest distance) from the priority queue and greedily expand the already fixed pairs with this new pair. Usually, the newly fixed match will introduce further unexplored neighbors, such that the algorithm continues from step ii), while keeping track of already fixed basic blocks.

The algorithm terminates if the priority queue is empty, i. e., if no further neighbors can be explored. After terminating the search, the algorithm computes the distance between signature and target program by adding the TEDs for each fixed block pair. If basic blocks from the signature could not be matched (e. g., if the signature is larger than the target program) it adds the TED between unmatched blocks and an empty tree to the overall distance.

Our neighborhood exploration algorithm is greedy and avoids expensive backtracking steps. Given that we apply the algorithm for all pairs of starting point/matching candidate, there is a high chance to find reasonable matches between signatures and target programs. Our neighborhood exploration results in a sorted list of distances, revealing the code parts most similar to the signature, i. e., the bug.

4. IMPLEMENTATION

We now describe the implementation details that we have left out in the previous section for brevity reasons.

4.1 Program Preprocessing

TEDEM's preprocessing phase has two distinct phases, namely the *Disassembly and Structuring* (Section 4.1.1) and the *Semantic extraction* (Section 4.1.2).

4.1.1 Disassembly and Structuring

We use the Interactive Disassembler (IDA) to extract the disassembly and basic block structure from the programs. IDA supports many operating systems (e. g., Windows, Linux, Mac OS X). We chose to focus on disassembling x86 binaries given their high popularity. We make assumptions about data structures (e. g., register names and their widths), but with some engineering effort, our system could also be applied to x86-64 binaries.

Using IDA, for each (strict) basic block, we export the start address, the instructions, and record which function it belongs to. In addition, for each basic block, we use IDA to export the Control Flow Graph (CFG) to keep track of a list of succeeding and preceding basic blocks.

Note that we assume that binary programs are not obfuscated. Our use case is finding bugs in commodity software, which usually is not obfuscated. For software developers, there is also no real benefit to evade our system, so we think this is a reasonable assumption. Clearly, our approach fails for any obfuscated software, as obfuscation destroys most of the CFG and the syntactical information. However, we consider (typically obfuscated) malware out of scope, because we primarily aim to secure legitimate software by finding locations that should be patched.

4.1.2 *Semantic Extraction*

We then transform the output of IDA into the intermediate representation (IR) that is part of METASM [9]. We chose METASM because it can symbolically accumulate assembly instructions in a compact and reasonably accurate form. In order to capture the effect of a basic block on the state of a program, namely registers, memory, branch conditions an successors, we use METASM's capability to accumulate the computational steps of a given sequence of assembler instructions in the form of compact equations. An equation may look like this: $eax := Ind(4, esi + edx * 4) + ebx * 2$, where $Ind(x, y)$ refers to the x-byte value at memory address y. These equations map the basic block's input state to its output state, condensing its semantic effect.

METASM uses typed equations which allow symbolic simplifications. We use easy simplifications such as constant folding, but also perform more complicated steps, like accumulating arithmetic/bitwise operations, to simplify, shorten, and normalize the expression. We further support machine-specific idioms such as `xor eax, eax` in order to reduce bogus dependencies on previous computations. In this concrete example, we model that `eax` becomes zero and we are not interested in the previous values of the register (although they, in principle, are used by `xor`).

We then convert these equations into S-Expressions. An S-Expression is a notation for a tree-like data structure, where a tree is noted as a bracketed, whitespace-separated list of its children. A child may be bracketed itself, denoting that it is a sub tree. For example, the equation above as an S-Expression is represented as
$(:= \ eax \ (+ \ (Ind \ 4 \ (+ \ esi \ (* \ edx \ 4))) \ (* \ ebx \ 2)))$.

4.2 Signature Matching

The actual signature matching consists of two separate steps. First, we find candidate basic blocks in the target program, which are very similar to a distinctive block from the signature (Section 4.2.1). Second, we explore the matches' neighborhoods for other good matches (Section 4.2.2).

4.2.1 *Basic Block Comparison*

One of our core ideas is using a TED to compare two basic block based on their lists of equations. We explored several notions of tree edit distances. The basic edit distance algorithm for trees performs node-wise operations only.

Tekli et al. [26] not only pre-compute costs for operations on subtrees with the dynamic programming approach used for classic string edit distances, but also add a notion of subtree commonality, which allows insertion/deletion of formerly adjusted subtrees. We chose this algorithm, as it recognizes and utilizes similarity in subtrees at a runtime of $\mathcal{O}(n^2)$, where n is the number of tree nodes. In the context of our work, tree nodes are subexpressions, i. e., operands and operations of an equation.

We ported the TED as proposed by Tekli et al. to C++. Figure 2 shows an example of the TED: In order to transform the left tree to the right tree, the TED first has to replace the left-most operand (`eax` by `edx`) and delete an operand ("2"). The power of the algorithm is demonstrated by the last edit operation, in which the common subtree (dashed square) replaces an operator ("+"). An algorithm with only node-wise manipulations would need to perform multiple single steps to move the subtree.

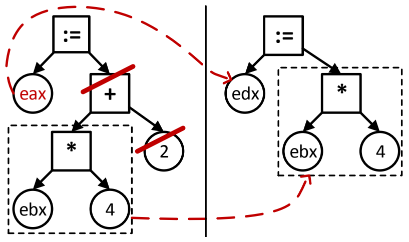

Figure 2: Exemplary tree edit distance with subtree-edits

As mentioned before, the equations that define a basic block's behavior are independent from one another. Because of the explicit computation order defined by the tree structure, the same is true for subexpressions. This allows reordering for equations and also for subexpressions (at least if the operation is commutative). We define an arbitrary but deterministic order of the elements and sort the trees accordingly, such that we avoid edit operations for basic blocks that only differ in the sequence of equations or elements.

4.2.2 *Neighborhood Exploration*

Based on these basic block-wise distances, we implemented the algorithm to match the bug signature (as described in Section 3.2.4). This section describes important performance impacts of the concrete implementation.

First, the *matching algorithm* finds a mapping between basic blocks to minimize the sum of all mapped distances. The implemented *Hungarian Method* [6] has a runtime of $\mathcal{O}(n^3)$ to match n nodes. Because we are only matching neighbors of one basic block from the signature with the neighbors of its candidate match in the target program, n is quite small — usually not larger than two. Apart from CFG considerations, this shows once more how important it is to operate the matching algorithm locally.

Instead, the computation of the basic block-wise edit distance dominates the runtime, especially when searching for promising starting points for the neighborhood exploration. Thus, we implemented a parallel search for starting points and matching candidates. In principle, also the other parts of our approach can run in parallel. For example, the signature matching algorithm can trivially be parallelized, as the metric splits up independently at the level of basic blocks — which we leave open to future work.

5. EVALUATION

We implemented a prototype of TEDEM that we evaluated on a variety of real-world bug examples. We group our test cases into three scenarios: (i) The identification of *shared code bugs* between related programs sharing one codebase; (ii) The identification of *fork bugs* between different programs sharing a (partly) similar codebase; (iii) The identification of *copy'n'paste bugs* in a single program.

To build our test cases, we first identified known vulnerabilities in binary software using IDA and *BinDiff*. We relied on these tools at this point because the initial identification of patched locations is out of scope for our work. We then manually created a signature for each test case. In most cases, using the entire vulnerable function as "easy" signature produced insightful and precise results. Such a signa-

App.	Bug/Patch	Sign.	Target	Rank #	
Pidgin	CVE-2013-6484	51/ 51	Adium	110/ 6,019	
		11/ 51		1/ 6,019	
Adium		42/ 42	Pidgin	18/ 3,965	
		11/ 42		1/ 3,965	
Pidgin	CVE-2013-6485	131/131	Adium	1/ 6,019	
Word 2010	MS13-072	59/ 59	Word 2003	1/28,564	
			Word 2007	1/44,626	
			Word 2013	1/21,624	
			Comp. Tool	1/13,035	
			Word Viewer	1/16,613	
PuTTY	CVE-2011-4607	980/980	PuTTYcyg	1/ 1,382	
			TuTTY	1/ 1,651	
PuTTY	CVE-2013-4208	37/ 37	PuTTYcyg	12/ 1,382	
			TuTTY	1/ 1,651	
		33/ 33	PuTTYcyg	4/ 1,382	
			TuTTY	1/ 1,651	
ImageM.	CVE-2009-1882	613/613	GraphicsM.	118/ 2,815	‡
ImageM.		1/613	GraphicsM.	1/ 2,815	‡
GraphicsM.		210/210	ImageM.	1/ 3,299	
Pidgin	CVE-2011-4601	5/ 19	Pidgin	1/ 1,242	†
ProFTPD	Backdoor	4/336	ProFTPD	4/ 1,711	
		3/336		1/ 1,711	

Table 1: Overview of evaluation results. The size of signatures is given in the form <*number used BBs/total BBs in vuln. function*>. The rankings marked with † and ‡ are further explained in Sections 5.2.2 and 5.3.1 respectively.

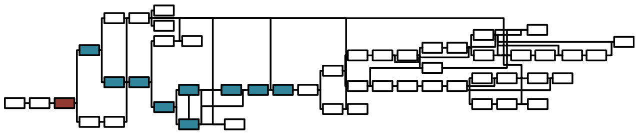

Figure 3: CFG (left to right, strict basic blocks) of `reply_cb()` in `Pidgin/libpurple` v2.10.7. The signature used to search for CVE-2013-6484 is highlighted. The patched block is highlighted on the far left.

ture can in certain cases be too broad, though, as it contains *noise* unrelated to the vulnerability, such as generic function prologues and epilogues that vary between different compilers and settings. Especially for smaller functions, signatures spanning entire functions can thus be too noisy/generic.

If not explicitly stated differently in the following, we used entire functions as signatures in our experiments. We chose to use entire functions as then the manual effort to create such signature is minimal — the analyst only needs to know the name/address of the vulnerable function. In principle, more fine-grained signatures obtain more accurate results. However, defining "good" signatures is a subjective process. Our choice to match entire functions allows for better comparability and repeatability of our work and guarantees sound experiments. We also perform experiments with fine-grained signature to show their strengths.

The evaluation results are summarized in Table 1 and we give detailed descriptions of the experiments in the following.

5.1 Shared Code Bugs

A typical use case for finding bug doublets arises when source code is shared among multiple software projects. Thus, we evaluate two such use cases in this section.

5.1.1 *Libpurple*

`Pidgin` v2.10.8 (2014-01-28) closes several vulnerabilities in libpurple. Among these are CVE-2013-6484 and CVE-2013-6485, two unrelated vulnerabilities that allow a remote attacker to crash a running Pidgin instance due to insufficient input verification. We used TEDEM to search for these two vulnerabilities in the most recent version of `Adium` (v1.5.9), a popular messenger for `Mac OS X` that also ships libpurple in compiled form. We manually confirmed that Adium still suffered from the two vulnerabilities that had already been patched in `Pidgin`.

For CVE-2013-6484, we first used the whole vulnerable function (51 basic blocks) in `Pidgin` as signature to search for the corresponding function in `Adium` (42 basic blocks). The size difference of the two functions resulted from `Pidgin` inlining a function. Given this coarse signature, TEDEM ranked the corresponding function in `Adium` (42 basic blocks) #18. On a relative scale this means that the analyst needs to manually inspect less than 0.5% of the functions in order to find the vulnerable code part. Next, we manually selected a signature of 11 basic blocks from the 51 basic blocks of the vulnerable function in `Pidgin` as depicted in Figure 3: Starting with the basic block whose branch condition was patched, a handful of succeeding basic blocks are selected, which validate network-received data. For this more fine-grained signature TEDEM ranked the corresponding function in `Adium` even #1.

For CVE-2013-6485, we used the entire unpatched function (131 basic blocks) from `Pidgin` as signature and unambiguously identified the corresponding function in `Adium` (136 basic blocks) at rank #1.

We also examined the other functions that TEDEM ranked high. We found that many of the functions were comparably exposed parsing network data. E. g., for CVE-2013-6485, a function that was previously reported to suffer from a similar memory corruption vulnerability[1] ranked #6.

This example across OS-boundaries also highlights the robustness of our approach in comparison to tools that largely rely on a coherent static call graph like *BinDiff*: When comparing the Windows and Mac versions of libpurple, *BinDiff* (v4.0.1) fails to correctly match the vulnerable functions to their counterparts. This is probably due to both functions being callbacks with no parents in the static call graph. However, even though our approach itself is not specific for a single OS, a particular bug signature may be very well.

5.1.2 *Microsoft Word*

Microsoft patched 13 vulnerabilities in various versions of `Word` for Windows[2]. We manually identified one of the patched locations in `Word 2010`, the most recent affected application. In `WRD12CNV.DLL` (5.21 MB) an additional boundary check on an array-index was introduced inside a loop in a certain function (virtual address `319421E7h`, 59 basic blocks). Apparently, this check tackles a memory corruption vulnerability. Given the patched function in `Word 2010` as signature, TEDEM unambiguously identified the equivalent function in all other patched applications (rank #1): `Word 2003` (`WINWORD.EXE`, 11.7 MB), `Word 2007` (`WWLIB.DLL`, 17.2 MB), `Office Compatibility Pack` (`WRD12CNV.DLL`, 4.46 MB), and `Word Viewer` (`WORDVIEW.EXE`, 8.4 MB). Also, TEDEM correctly highlighted an interesting circumstance: All other applications' *patched* versions were more similar to `Word 2010` *unpatched* than to `Word 2010` *patched*. Indeed, we could manually confirm that the other applications had been patched differently than `Word 2010`. Checks were added that were already present in `Word 2010` *unpatched*. The

[1]`http://www.pidgin.im/news/security/?id=34`
[2]MS13-072: `https://technet.microsoft.com/library/security/ms13-072`

array-index boundary check introduced in `Word 2010` *patched* was still missing in the *patched* other applications, though. The identification of this possibly critical difference in patch levels is an exemplary use case for TEDEM. Lastly, using the same signature, we could also pinpoint the corresponding function in a more recent version of `Word 2013` (`WRD12CNV.DLL`, 6.01 MB). We found the same checks to be present as in `Word 2010` *patched*.

We found that *BinDiff* repeatedly matched wrong functions between the different applications and between the different versions of the same application. For example, *BinDiff* could not identify the discussed patched function in `Word 2003`, neither through incremental version diffing nor through cross-application diffing with `Word 2010`.

5.2 Fork Bugs

When software projects are *forked*, it can be difficult to tell if a bug also affects forked projects. We evaluate TEDEM's ability to pinpoint bugs in two forked applications based on signatures extracted from their parent applications.

5.2.1 PuTTY

`PuTTY` is an open source SSH client for Windows. Prior to v0.62, passwords were not wiped from memory (CVE-2011-4607) and until v0.63 the same was the case for certain cryptographic data (CVE-2013-4208). We examined the `PuTTY` forks `TuTTY` v0.60.2.0 and `PuTTYcyg` r20101029 for the presence of these bugs using TEDEM.

Given the vulnerable function of CVE-2011-4607 (980 basic blocks) in `PuTTY` v0.61 as signature, TEDEM ranked the corresponding functions in `TuTTY` and `PuTTYcyg` #1. The bug described in CVE-2013-4208 is contained in two functions in `PuTTY` v0.61. Given these two functions as signatures, TEDEM ranked the corresponding functions #1 in `TuTTY`. For `PuTTYcyg`, we could only achieve ranks #12 and #4, because the CFG was heavily altered, e. g., due to `memcmp()` being inlined. However, these ranks are still in the top 1% and could thus be helpful to an analyst.

5.2.2 ImageMagick

`ImageMagick` is an open source image processing tool. Prior to v6.5.2-9, it suffered from an integer overflow bug in TIFF image processing code (CVE-2009-1882). The tool `GraphicsMagick` was forked from `ImageMagick` as early as 2002 and is still actively developed. We used TEDEM to check for the presence of the bug in `GraphicsMagick` v1.3.7.

Given the vulnerable function from `ImageMagick` v6.5.1-2 (613 basic blocks) as signature, TEDEM ranked the corresponding function in `GraphicsMagick` (210 basic blocks) only #118. This is due to a large and dominating function being inlined into `ImageMagick`'s vulnerable function. In turn, this inlined function's counterpart in `GraphicsMagick` comes #1 in the similarity ranking produced by TEDEM in the reverse direction. This function is (statically) called by only the vulnerable function in `GraphicsMagick`. Accordingly, it was easily possible to pinpoint the vulnerable function in `GraphicsMagick` given TEDEM's output. In general, this one-directional behavior of finding the bigger function with the inlined component, but not vice versa, is what one would expect in the face of inlining and shows that our approach tolerates inlining to some extent.

However, when we compile a signature of only a single basic block from `ImageMagick`, namely the one with the integer overflow, we can also rank the corresponding function in `GraphicsMagick` on #1.

Dist.	Function name
0.0	▨ receiveauthgrant
0.0	▨ parseicon
13.8	▨ receiveauthrequest
21.8	▨ receiveadded
23.7	▨ receiveauthreply
36.9	memrequest
98.8	▨ parseadd
120.8	msgack
150.3	▨ clientautoresp
176.8	oscar_buddy_menu
190.1	oscar_get_ex..._status
202.7	purple_bosrights
215.3	purple_odc_send_im
252.2	incomingim
260.9	infochange
284.0	oscar_send_im
292.5	find_acct
304.8	oscar_status_types
382.2	oscar_actions
395.2	▨ parsemod
399.6	▨ aim_info_extract
451.9	aim_request_login
566.0	infochange
571.1	oscar_..._display_icq
592.5	oscar_auth_sendrequest
596.3	aim_putsnac
599.4	purple_ssi_authreply
635.0	oscar_...and_status

Table 2: Functions in `Pidgin/libpurple` with the lowest distance to our signature extracted from CVE-2011-4601. Marked functions lack the necessary check.

5.3 Copy'n'Paste Bugs

Another source for bug doublets is caused by developers that adopt faulty source code and miss to patch it in case of upstream bugs. We evaluate TEDEM for two such cases.

5.3.1 Libpurple

We evaluated the same vulnerability (CVE-2011-4601) in `Pidgin`'s libpurple on binary level as Yamaguchi et al. did on source code level [28]: Prior to v2.10.0, a missing check in the function `receiveauthgrant()` enabled remote attackers to crash `Pidgin`. Yamaguchi et al. identified nine other functions suffering from the same vulnerability.

We compiled `Pidgin` v2.10.0 on Linux with default settings, but without optimization, because otherwise several vulnerable functions were inlined into one particular function. This would have made results incomparable. To exclude obvious boilerplate code, we chose only the immediate neighbor-blocks to the actual vulnerability, ending up with a signature of 5 of the 19 basic blocks in the original vulnerable function `receiveauthgrant()`. Table 2 lists those functions from the roughly 2360 functions in the binary libpurple that TEDEM identified as most similar to the signature. Indeed Table 2 resembles very much the listing reported by Yamaguchi et al. for their tool. For example, they also found 9 among the thirty best ranked functions to lack the necessary check. Interestingly, we found three functions that Yamaguchi et al. did not report on and vice versa. Overall, this demonstrates the ability of TEDEM to compete with source code based tools in terms of quality of results.

5.3.2 ProFTPD (Backdoor)

In a variant of `ProFTPD` v1.3.3c, a backdoor was found in a handler function for the *HELP* FTP command. The backdoor gave privileged system access to anyone issuing the command with a certain argument. We constructed an artificial use case from this incident by installing a similar backdoor in the handler of the *MKD* FTP command. Using the original backdoor (4 basic blocks) as signature, we used TEDEM look for the other backdoor in a `ProFTPD` binary compiled on Linux with default settings. TEDEM ranked the manipulated handler for *MKD* (78 basic blocks) #4. The manipulated handler ranked #1 for a more fine-grained signature composed of 3 basic blocks. This example again highlights that matching based on fine-grained signa-

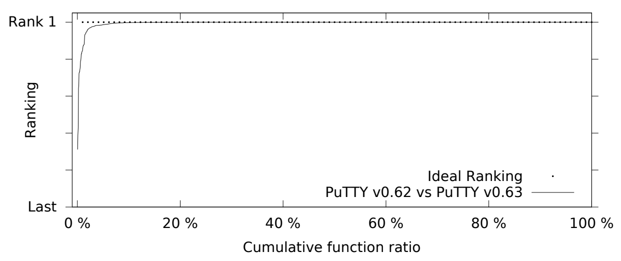

Figure 4: Rankings of function-level matching.

tures can in our use case often be more powerful than simple function-level matching.

5.4 Systematic Function-Level Matching

Lastly, we measure the overall applicability of our approach to match functions. Although function matching is not our considered use case, such a systematic experiment allows to illustrate the generality of our method. We chose to compare two similar binaries for which we have a ground truth of functions that should be mapped. In particular, we compare the binaries of two subsequent PuTTY versions (v0.62 and v0.63) that differ only slightly. We use their symbolic information to create the ground truth of functions that should be mapped. We do not use symbolic information as part of our algorithm, i.e., we use stripped binaries.

Figure 4 shows a CDF of the ranks when mapping the functions in PuTTY v0.62 in v0.63. Our experiments show that 77.8% of the functions were correctly mapped (rank #1). Only 6% of the functions ranked badly (rank > 20). We manually inspected the reasons for these non-optimal rankings and found that some of them were caused by slight modifications in the two PuTTY versions. Others were caused by functions that substantially changed their CFG, e. g., functions that had significantly more basic blocks than their counterpart. However, overall, the experiment shows that our method generally produces reliable matches and works beyond the concrete examples discussed above.

6. DISCUSSION AND FUTURE WORK

TEDEM can leverage semantic signatures on a binary level to find similar bugs in other programs or other functions within the same program. In this section we discuss limitations and future work directions of our work.

False Positives. Our distance metric can compare arbitrary divergent code. If the target program does not contain a vulnerable function, our approach still produces a ranking of functions with the distance to the signature. However, the distances are significantly larger than the distances of similar code. We will investigate if there are reasonable mechanisms to determine meaningful thresholds that can filter out false positives in future work.

Robustness. Although the evaluation has shown that our approach works well in most practical examples, we have to acknowledge our method's sensitivity to changes in binary code. This section discusses many other typical binary code modifications that our system design tolerates up to some degree. However, we exclude obfuscated binaries from the scope of our method explicitly.

Using a TED-based similarity metric allows to compare code even if it slightly changed. For example, although register renaming or register spilling would slightly increase the TED, the overall similarity remains high. This is also true if the source code was slightly modified (e.g., inserted instructions, change of constants, global variables, etc.). We can also deal with instruction reordering, as we use a deterministic ordering of the equations and their sub-expressions. Lastly, also function inlining is not problematic, because we dynamically broaden a match and thus can also match subgraphs (and not only entire functions). This increased robustness against syntactical changes is thanks to our accumulation and normalization into semantic representations.

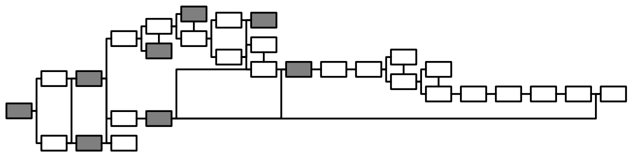

Figure 5: CFG (strict basic blocks, left to right) of *Aurora bug* constructor. Good matches from the copy constructor are highlighted. They are obviously not connected densely.

However, our bug search requires that some neighborhood properties are preserved in the CFG. For example, we found that CVE-2010-0249, the bug used in *Operation Aurora*, violates this requirement. The vulnerability was a *use-after-free* bug in Internet Explorer versions 6–8 that allowed remote code execution. A constructor and copy constructor of a C++ class were patched in a similar way to correctly update reference counters in other objects. The regular constructor is far larger than the copy constructor (26 vs. 8 basic blocks). Figure 5 shows that matching basic blocks are hardly connected to each other, which causes the neighborhood exploration algorithm to miss the constructor given the unpatched copy constructor as bug signature.

Many of the aforementioned syntactical changes can be induced by typical optimization strategies of compilers. However, note that this is also only relevant if the optimization strategy differs between signature and target program.

Fine-Tuning Bug Signatures. Our experiments are based on function-level signatures. As we have shown, fine-granular bug signatures can be much more accurate for identifying buggy code locations. As such, defining more specific bug signatures is a powerful way to boost the accuracy.

Another possible way to increase the accuracy is assigning weights to individual parts of the signature. Weights allow us to respect gradual importance of specific structures, e. g., giving the change in a code location constant or the use of a different register less importance than, say, changing the operand of a computation.

Lastly, creating bug signatures could be automated. For example, an analyst could flag vulnerable source code parts, compile the code, and use debug information (i.e., source code location information) to map the vulnerable source code to its binary code representation.

Finding Patched Code. Typically, patched code is quite similar to unpatched code. In general, it is not trivial to determine if the functionality that we find to be similar to our signature is actually vulnerable. As of now, we cannot determine if the conditions allowing an exploitation are satisfiable. However, first experiments have shown that we can define an additional signature spanning the patched code, which we can use for false positive testing. Intuitively, the patched signature matches more closely to patched binaries than to unpatched ones. By comparing both similarity scores it is thus possible to distinguish patched from unpatched binary code, as shown for `MS Word` (Section 5.1.2).

Signature	Target	Signature Size	Candidate Search	Neighborh. Exploration
ProFTPD	ProFTPD	3 BBs	1.5 s	0.02 s
PuTTY	TuTTY	14 BBs	11.6 s	1.12 s
PuTTY	PuTTYcyg	13 BBs	11.4 s	1.32 s
Libpurple	Adium	51 BBs	349.7 s	7.95 s
Word 2010	Word 2013	58 BBs	1043.1 s	34.32 s
GraphicsM.	ImageM.	210 BBs	1332.9 s	116.61 s
PuTTY	TuTTY	978 BBs	2482.5 s	1958.13 s

Table 3: Runtimes of signature searches

Scalability. The size of the signature influences runtime linearly. For our targeted use case of fine-grained search with small signatures, we achieve reasonable runtimes of less than a minute (see Table 3) on a workstation with Intel Core i7-2640M @ 2.8GHz with 8GB DDR3-RAM. Clearly, the search for initial candidate matches dominates the runtime. For larger signatures, like in the extreme case of `PuTTY` (978 basic blocks), the runtime rises to about 42 min. However, 33 min have to be accounted to the candidate search, as many pairwise basic block distance computations have to be performed. In contrast, the neighborhood exploration algorithm has a relatively low overall runtime, although it is invoked very often due to a high number of starting points and their matching candidates. For example, in `PuTTY`, the algorithm had to be invoked 19,560 times (978 basic blocks times $n = 20$ matching candidates), but the runtime added up to 6.5 min with an average of 20 ms per invocation.

The size of the target program influences the runtime of the candidate search linearly. However, as we have seen, e.g., for `MS Word`, TEDEM is still scalable even on commodity user hardware. In future work we will look into methods how to further improve the performance. For example, an M-Tree, a data structure for fast k-nearest neighbor search in metric spaces [8], could speed up the candidate search.

7. CONCLUSIONS

We have shown that re-finding similar bugs on the level of binary code is possible. By this we enhance existing research that aims to find similar code bugs on the source code level. Once a bug is known to an analyst, our concept of semantic signatures can be applied to search for similar bugs in the same binary or even in different programs. We searched for various real-world bugs (caused by forks, shared code, or copy'n'paste errors) with promising accuracy.

Acknowledgements

This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16BP12302, EUREKA-Project *SASER* and by the German Research Foundation (DFG) research training group *UbiCrypt* (GRK 1817).

8. REFERENCES

[1] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using Spatio-temporal Information in API Calls with Machine Learning Algorithms for Malware Detection. In *AISec '09*.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *NDSS '11*.

[3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *IEEE S&P*, 2012.

[4] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *ASPLOS XVI*, 2011.

[5] T. Dullien and R. Rolles. Graph-based Comparison of Executable Objects. *SSTIC*, 5, 2005.

[6] A. Frank. On Kuhn's Hungarian Method – A Tribute from Hungary. Technical report, Oct. 2004.

[7] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Info. and Comm. Security*. 2008.

[8] F. Gauthier, T. Lavoie, and Merlo. Uncovering Access Control Weaknesses and Flaws with Security-discordant Software Clones. In *ACSAC '13*.

[9] Y. Guillot and A. Gazet. Automatic Binary Deobfuscation. *Journal in Comp. Virology*, 2010.

[10] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Sec. '13*.

[11] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *IEEE S&P*, 2012.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *ICSE*, 2007.

[13] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary Function Clustering Using Semantic Hashes. In *IEEE ICMLA 2012*.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 28(7), July 2002.

[15] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Exec. In *RAID'05*.

[16] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *ACM PPREW 2013*.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in OS Code. In *USENIX OSDI '04*.

[18] C. McMillan, M. Grechanik, and Poshyvanyk. Detecting Similar Software Applications. In *ICSE '12*.

[19] J. Ming, M. Pan, and D. Gao. iBinHunt: Binary Hunting with Inter-Proced. Ctrl Flow. In *ICISC'12*.

[20] G. Myles and C. Collberg. K-gram Based Software Birthmarks. In *ACM SAC '05*.

[21] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting Vulnerable Software Components. In *ACM CCS 2007*.

[22] J. Newsome and D. Song. DTA for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS '05*.

[23] B. H. Ng and A. Prakash. Expose: Discovering Potential Binary Code Re-use. In *COMPSAC 2013*.

[24] P. Oehlert. Violating Assumptions With Fuzzing. *IEEE S&P '05*.

[25] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE S&P '10*.

[26] J. Tekli, R. Chbeir, and K. Yetongnon. Efficient XML Structural Similarity Detection using Sub-tree Commonalities. In *SBBD*, 2007.

[27] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *ACM SOSP 2013*.

[28] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *ACSAC 2012*.